Published in final edited form as:

Concurr Comput. 2014 December 1; 26(18): 2836–2855. doi:10.1002/cpe.3181.

Massively Multithreaded Maxflow for Image Segmentation on the Cray XMT-2

Shahid H. Bokhari^{1,*}, Ümit V. Çatalyürek², and Metin N. Gurcan³

Shahid H. Bokhari: shb@acm.org; Ümit V. Çatalyürek: catalyurek.1@osu.edu; Metin N. Gurcan: metin.gurcan@osumc.edu

¹Algopath LLC, 20660 Stevens Creek Blvd. #164, Cupertino, CA, 95014

²Departments of Biomedical Informatics, and Electrical & Computer Engineering, The Ohio State University, Columbus, Ohio 43210

³Department of Biomedical Informatics, The Ohio State University, Columbus, Ohio 43210

SUMMARY

Image segmentation is a very important step in the computerized analysis of digital images. The maxflow mincut approach has been successfully used to obtain minimum energy segmentations of images in many fields. Classical algorithms for maxflow in networks do not directly lend themselves to efficient parallel implementations on contemporary parallel processors. We present the results of an implementation of Goldberg-Tarjan preflow-push algorithm on the Cray XMT-2 massively multithreaded supercomputer. This machine has hardware support for 128 threads in each physical processor, a uniformly accessible shared memory of up to 4 TB and hardware synchronization for *each* 64 bit word. It is thus well-suited to the parallelization of graph theoretic algorithms, such as preflow-push. We describe the implementation of the preflow-push code on the XMT-2 and present the results of timing experiments on a series of synthetically generated as well as real images. Our results indicate very good performance on large images and pave the way for practical applications of this machine architecture for image analysis in a production setting. The largest images we have run are 32000² pixels in size, which are well beyond the largest previously reported in the literature.

Keywords

Cray XMT-2; maxflow; image segmentation; multithreading; massively parallel processing

1. INTRODUCTION

The automated analysis of digital images is an important problem in many fields such as image understanding and computer vision. In the biomedical field for example, developments in imaging technologies have led to a flood of data in the form of digitized pathology slides, X-rays, CAT scans, MRI scans, etc. The analysis of these digitized images by human experts is time consuming, expensive and, more importantly, prone to intra- and inter-reader variability. In recent years there has been great success in development of

*Correspondence to: Algopath LLC, 20660 Stevens Creek Blvd. #164, Cupertino, CA, 95014. shb@acm.org.

computer-assisted image analysis systems for both radiological and pathological images. These systems learn from human experts, the current knowledge-base and large databases. They are not intended to replace human readers but to provide feedback to the readers to minimize errors both in detection and diagnosis stages [1, 2, 3, 4, 5, 6, 7].

Automated analysis requires very heavy consumption of computation power. At the same time, increasing resolution of microscopic image scanners, satellite/aerial sensors and gigapixel cameras is yielding very large images that are increasingly difficult to analyze on conventional uniprocessor machines. There has been some success in partitioning images over the processors of a distributed memory system and carrying out operations on each image tile with intervening communication steps. This type of parallelization cannot take into account large features that might overlap several tiles and require a prohibitive overhead of interprocessor communication for correct analysis.

In the present paper we describe an implementation of an image processing operation on the Cray XMT-2 massively multithreaded supercomputer [8], which is the latest version of an architecture that started with the Tera [9, 10]. This machine has a huge uniformly accessible shared memory (up to 4 TB in currently available implementations), hardware support for 128 threads per processor and *individually* lockable 64-bit memory locations. It is thus well suited to large-scale, fine-grained algorithms of the type that are required in image processing.

We specifically address the problem of image segmentation, which is a fundamental issue in image analysis. There are various approaches to image segmentation—one of the most successful uses maximum flow. Grieg et al. [11] published the earliest paper on this approach in 1989. In recent years, Boykov et al. have been active in this area [12, 13, 14]. Jamriska et al. [15] show how the Boykov-Kolmogorov [14] algorithm can be improved by giving due consideration to cache locality. This consideration does not arise on the Cray XMT-2 as it obtains its speedup without caches.

In the maxflow approach, the image segmentation problem is converted into a maximum flow problem in which the minimum weight cut delineates the required segments of the image as explained in Fig. 1. This figure shows how a graph is created with nodes corresponding to each pixel in the image. Edges connect each node to its 4 nearest neighbors. Edges between nodes have weights inversely proportional to the difference in intensities (or colors) of the corresponding pixels. Thus edges between similar pixels have high weights, while edges between dissimilar pixels have low weights.

A source node is connected to one or more nodes known to be within the region of interest, with infinite weight edges. Similarly, one or more nodes outside the region of interest are connected to the sink with infinite weight edges. Segmentation requires us to find the minimum weight boundary around the region of interest. This is done by applying a maximum flow algorithm between the source and the sink. The minimum cut (a minimal set of edges that when removed from the graph, disconnects the source from the sink) marks the minimum cost boundary of the region of interest. In Fig. 1 the segmentation is indicated by blue arrows.

The connection of the source (sink) to the grid is often done manually, typically by identifying one or more disk that are unambiguously within (outside) the regions of interest. This can also be done automatically by specifying ranges of colors that lie unambiguously within or outside these regions. In either case, the burden is on the flow algorithm to start with these specified “seed” regions and find the minimum weight cut that separates the region of interest from the rest of the image.

Traditional maximum flow algorithms are difficult to parallelize on conventional parallel machines. For example, one of the best performing algorithm for maxflow, namely Goldberg and Tarjan’s preflow-push [16], has not met with much success in the context of parallelization despite having an intrinsically parallel structure. In fact many parallel implementations are hard pressed to match the performance of Goldberg’s HiPR preflow-push code (www.avglab.com/andrew/soft.html) on a modest serial machine.

We describe our implementation of the standard preflow-push algorithm on the Cray XMT-2 for rectangular grid graphs (of the type generated by image segmentation problems). Preflow-push requires the use of heuristic steps to speed up solution. Of these, the global relabeling heuristic is most beneficial in terms of reducing solution time but, at the same time, most expensive in terms of overhead. This is because the breadth first search required by global relabeling is difficult to parallelize as it requires a shared parallel queue. However, the primitive operations available on the Cray XMT-2 allow simple implementations of queues to perform well. We use a standard implementation for queues developed by Gottlieb et al. [17], in conjunction with an adaptive method of scheduling this operation to obtain a very efficient implementation.

2. MAXIMUM FLOW

A classical problem in graph theory is that of finding the maximum flow. Here we are given a weighted graph (with n nodes and m edges) in which the weights represent edge capacities. There are two distinguished nodes named source and sink and the problem is to find the maximum flow from the source to the sink. The flow in each edge must be less than or equal to its capacity. Conservation of flow must apply at all nodes, i.e., flow coming into a node must equal the flow going out (this constraint does not apply to the source and sink).

A famous theorem by Ford and Fulkerson [18] states that the maximum flow in a network equals the minimum cut. Here the cut is a minimal set of edges that, when removed from the graph, will disconnect the source from the sink. In order to find the minimum cut in a flow graph we need to find the maximum flow which then yields the minimum cut.

There is a long history of algorithm development for the maximum flow problem, starting with the original work by Ford and Fulkerson [18]. Their algorithm is not polynomial in the strict sense, since examples can be contrived where it takes time proportional to the magnitude of the weights on the edges rather than to a polynomial in the number of edges. The earliest polynomial time algorithms are due to Edmonds and Karp ($O(nm^2)$) [19], Dinitz ($O(n^2m)$) [20], and Karzanov ($O(n^3)$) [21]. A survey of these and later algorithms is available in [22]. One of the most successful algorithms for which an efficient serial implementation

is openly available and, at the same time is intrinsically parallel in nature, is the Goldberg-Tarjan preflow-push algorithm ($O(n^3)$) [16].

Earlier maximum flow algorithms started with a legal flow (i.e., one that satisfies capacity and conservation constraints) and then applied a sequence of augmentation steps, while maintaining legality. This approach is difficult to parallelize. The preflow-push algorithms of Goldberg et al. (and their variants) do not maintain legal flow. Rather, the conservation condition is violated during the course of the algorithm and is corrected at the end. In principle, these algorithms are straightforward to parallelize but in practice many heuristics are required for good performance.

Goldberg's HiPR code is generally considered one of the fastest openly available maxflow implementations (www.avglab.com/andrew/soft.html). The current version is 3.7 and was released on 9th February 2011. As we shall show below, the preflow-push algorithm is inherently parallel and is thus our choice for implementation on the Cray XMT-2. Recent work by Goldberg et al. has led to the development of the *Incremental Breadth First Search* algorithm [23] which is faster and more robust than the Boykov-Kolmogorov algorithm [14] for vision problems. However it is not clear whether this new algorithm can be parallelized for massive problems. We therefore utilize the inherently parallel preflow-push algorithm.

2.1. The preflow-push algorithm

The following is a condensed exposition of the preflow-push algorithm from Goldberg and Tarjan [16].

2.1.1. Definitions

- We are given a weighted graph $G = (V, E)$ with $n = |V|$ nodes.
- There are two distinguished nodes s (source) and t (sink).
- Each edge (u, v) in E has a capacity $c(u, v)$ and flow $f(u, v)$.
- A *flow* in this graph obeys the usual capacity and conservation constraints.
- A *preflow* obeys capacity constraints, but the inflow into a node v may be greater than the outflow. The difference is called the *excess*, denoted $e(v)$.
- The *residual capacity* of an edge is $r_f(u, v) = c(u, v) - f(u, v)$.
- (u, v) is called a *residual edge* if $r_f(u, v) > 0$.
- The residual graph G_f is the subgraph induced on G by all nodes V and all residual edges.
- Each node is labeled with a non-negative *distance* label: $d(s) = n$, $d(t) = 0$ and $d(u) = d(v) + 1$ for every residual edge (u, v) .
- If $d(v) < n$ then $d(v)$ is a lower bound on the distance from v to t in G_f .
- If $d(v) = n$ then $d(v) - n$ is a lower bound on the distance from v to s in G_f .
- A vertex v (other than s or t) is called *active* if its excess is nonzero.

2.1.2. The Algorithm

```

initialize preflow:
  for each  $(s, u)$  set flow  $f(s, u) = c(s, u)$ ,
  all other edges have zero flow.
initialize labeling:
   $d(s) = n$  and
   $d(u) = 0$  for all remaining nodes.
repeat
  push all  $(u, v)$ :
    if ( $u$  is active,  $r_f(u, v) > 0$ 
    and  $d(u) = d(v) + 1$ ) {
       $p = \min(e(u), r_f(u, v))$ 
       $f(u, v) += p$ ;  $e(u) -= p$ 
       $f(v, u) -= p$ ;  $e(v) += p$ 
    }
  relabel all  $(u)$ :
    if ( $u$  is active and for each neighbor  $v$ 
    in  $G_f$ ,  $d(u) > d(v) + 1$ ) {
       $d(u) = \min_{\text{all neighbors}}(d(v) + 1)$ 
    }
until (active nodes = 0)

```

As we shall describe in Section 3.1, the fine grained synchronization hardware of the Cray XMT-2 allows the push and relabel operations to be executed in parallel. However, we have found the performance of the parallel algorithm to be appallingly poor unless heuristics are used. The two main heuristics used to speed up the preflow-push algorithm are Global relabeling and Gap relabeling [24, 25].

These heuristics are normally applied periodically and the decision when to apply these can be crucial for obtaining the best performance. In our parallel implementation of preflow-push we use global relabeling but not gap relabeling. Gap relabeling is only meaningful under the “Highest Level” selection strategy [24], which is difficult (if not impossible) to implement in a parallel context.

The repeat – until loop of the preflow-push algorithm includes global relabeling in the following manner.

```

repeat
  push all  $(u, v)$ ...
  relabel all  $(u)$ ...
  if ( $T_G()$ )
    global relabel all  $(u)$ 
until (active nodes = 0)

```

Here $T_G()$ is a function that determines whether a global relabel is to be applied. As we shall discuss in Section 3.4 below, this function typically returns true after every N passes through the loop, although we have developed a more refined approach.

2.2. Prior research in parallelizing maxflow

One of the earliest efforts at implementing parallel preflow-push was made by Alizadeh and Goldberg in 1992 [26]. This was on the Single Instruction Multiple Data Stream (SIMD) CM-2 Connection Machine.

In 1995, Anderson and Setubal [27] described a parallel implementation of the preflow-push algorithm on the Sequent Symmetry S81, an early shared-memory machine with 20 Intel 80386 processors. They reported little benefit in using the gap-relabel heuristic.

Bader and Sachdeva [28], in 2005, developed a “cache-aware” implementation of preflow-push on a Sun E4500 with 14 UltraSPARC II processors. Their implementation is carefully designed to exploit the cache memory of their target platform, a consideration that does not arise on the Cray XMT family, which does not have any cache. Interestingly, their research found the gap-relabel heuristic to be useful.

Our implementation differs from the above in that it is specifically designed for grid graphs of the type that arise in image segmentation problems. However, our success with this code indicates that extensions of this work to general graphs is likely to be successful.

Delong and Boykov [13] reported on a parallel algorithm for image segmentation that uses a modified “region push-relabel” approach and tested it on small numbers (1–8) of Intel Xeon processors. They pointed out the importance of running maxflow algorithms on graphs made up of large grids and state that the global heuristic steps are essential to obtain good performance. However, for the types of parallel systems they experimented with, these global relabeling steps limited scalability. They solved this problem by developing localized heuristic steps. However this approach requires the partitioning of the problem graph into regions, which in turn leads to issues of load balancing. Our implementation uses the standard global relabeling heuristic and does not require any partitioning and load-balancing because of the architectural features of the Cray XMT.

A publicly available parallel algorithm, `d_maxflow`, is described by Shekhovtsov and Hlavac [29]. This code is based on the Delong-Boykov approach discussed above, and suffers from the same issues of partitioning and load balancing. Furthermore, the shared-memory parallel version of this code is unable to compete with the original Delong-Boykov code [29, Table 3].

Another parallel code is `pmaxflow` available at code.google.com/p/pmaxflow/. This code has very limited performance. To quote from this website:

Results of our experiments show that *in some of the tests* our implementation reduces the solution time of the best sequential maximum flow solver (Goldberg’s solver). In particular, in on (sic) some graphs coming from vision area, *speed-ups of two can be obtained* (emphasis added).

Thus this code has very poor performance compared to our algorithm, which gives performance consistently superior to HiPR for all large problems.

Hong and He [30] have developed an asynchronous parallel version of the preflow-push algorithm and carried out a detailed investigation of its performance. This algorithm aims to improve parallel performance by eliminating the usage of locks and barriers. In addition, they develop an asynchronous global relabeling heuristic. They evaluate the performance of their code on large sets of graphs, but are able to obtain a speedup of only 3 over Goldberg's serial HiPR code for acyclic-dense graphs [30, Fig 1] of sizes 2000 and 4000 nodes. For other classes of graphs (of sizes as large as 500,000 nodes), they are barely able to match or exceed HiPR [30, Fig 2], or are unable to beat HiPR at all [30, Fig 3]. Our algorithm, on the other hand, consistently outperforms HiPR for large problems, as we shall demonstrate in Section 4.

3. IMPLEMENTATION

3.1. The Cray XMT-2

The Cray Extreme Multithreading (XMT-2) supercomputer is the second member of the XMT family of machines [8]. This architecture originated in the Tera [9, 10] and was refined into the MTA (Multithreaded Architecture) and MTA-2 [31, 32]. The MTA-2 was followed by the XMT (earlier code-named "Eldorado") [33, 34]. The performance of this machine is described in [35, 36, 37, 38].

While the XMT-2 has a *multithreaded* processor, called *Cray Threadstorm*, it is not a *multicore* processor like currently available commodity processors. Each processor, clocked only at 500Mhz, has 128 hardware thread-streams and one 21-stage instruction pipeline. Each thread-stream is equipped with 32 general purpose registers, 8 target registers, a status word, and a program counter, to store the state of a thread [8, 9, 10]. Therefore, each processor can maintain up to 128 software threads. In every cycle, a processor context switches among threads with ready instructions in a fair manner choosing one of the threads to issue its next instruction. A processor stalls only if no thread has a ready instruction. The number of threads utilized in any section of code is determined by the compiler based on counts of instructions in loops, expected overheads, etc. The compiler's decisions in this regard are usually very good and the programmer rarely has to specify the number of threads to use.

The memory system of the XMT is cache-less, and is globally accessible by all processors. The system supports a shared memory programming model. The global address space is built from physically distributed memory modules of 32 GB per processor, yielding a total memory size of 4 TB for a 128 processor system. Further, the address space is built using a hardware hashing mechanism that maps the data randomly to the memory modules, in blocks of size 64 bytes, to minimize conflicts and hot-spots. All memory words are 8 bytes wide and the memory is byte-addressable.

Programming the XMT-2 involves some code restructuring and the use of pragmas (compiler directives in code). Machine specific synchronization primitives can also be used. These are facilitated because there is:

1. Hardware support for 128 threads per processor,
2. Zero overhead thread switching,
3. A large uniformly accessible shared memory,
4. A Full/Empty synchronization mechanism on each 64-bit memory word, and
5. A powerful compiler and performance analysis tools[39].

The Cray XMT-2 permits extremely large images to be loaded into one globally shared memory to which all processors have uniform access. Each processor of the XMT-2 has hardware support for 128 threads, thus a 128 node machine could support 16,384 threads. This, in conjunction with hardware supported individually lockable memory locations, permits very fine grained parallelism to be applied to a problem. The computational operations possible on the XMT-2 are global and are not constrained to the neighborhood of a tile, as would be the case on a distributed memory machine. This allows us to implement parallel algorithms that would have been considered infeasible in the past.

While in the past considerable effort was required to implement maxflow on parallel computers, the task is relatively straight forward on the XMT-2 as we shall show in Section 3.2. The bulk of our time was taken up in exploring and fine tuning the heuristic part of the algorithm. We developed our code on the 16 processor ‘Egret’ XMT at Cray Inc. and the 128 processor ‘Cougar’ XMT at Pacific Northwest National Laboratory. The timings reported in this paper are from the 64 processor ‘Matterhorn’ XMT-2 at the Swiss National Supercomputing Centre, and on the 128 processor ‘Danriver’ XMT-2 at the Center for Applied High Performance Computing in Virginia. Our implementation is superior to the serial HiPR code beyond 2000² grids and we have run problems as large as 32000² grids.

3.2. Fine-grained synchronization

Each 64-bit word on the XMT-2 has an additional Full/Empty synchronization bit that permits it to be individually locked. The “wait until full, then read and set empty” instruction `val = readfe(address)` waits until the synchronization bit for address is full then *atomically* returns the value stored in address while setting the synchronization bit to empty. No other thread can subsequently read from address until the synchronization bit is set to full again. The “wait until empty, then write and set full” instruction `writeef(address, val)` waits until the synchronization bit for address is empty then writes val into it and *atomically* sets the bit to full. While these instructions are expressed as function calls, they are compiled into *single* machine instructions.

The following code fragment illustrates how these instructions are used to synchronize the parallel execution of the “push” operation of the preflow-push algorithm.

```
residual = node[u].capacity[v] - node[u].flow[v];
```



```

if((node[u].capacity[v]>0)&&(residual>0)){
    Excess = readfe(&node[u].excess);
    pushable = MIN(Excess, residual);
    temp = readfe(&node[u].flow[v]);
    temp = temp + pushable;
    writfe(&node[u].flow[v],temp);
    Excess = Excess-pushable;
    writfe(&node[u].excess,Excess);
    ...
}

```

Here, `node[u].capacity[v]` (`node[u].flow[v]`) is the capacity (flow) between node `u` and node `v`. The code pushes the excess from `u` to `v`, updating excess and flow as necessary (see Section 2.1.2). The use of **readfe** and **writfe** operations ensures that these updates are carried out consistently. Because of the power of these primitives, all nodes in the graph can be updated in parallel—the zero-overhead thread switching on the XMT-2 ensures that, should a node be found “empty,” execution switches *in hardware* to another available “full” node. As long as there is a pool of “full” nodes to be processed, computation proceeds without interruption. The challenge for the programmer is to ensure that such a pool is available during most of the lifetime of the program. Note how no essential modification is required to implement the maxflow algorithm on the XMT-2, other than use of the full-empty mechanism. This is but one example of how the architecture of the XMT family of machines permits fine-grained parallel computing in graph problems. Other examples appear in [37], [40], [41] and [42].

3.3. Synthetic Images

To carry out a comprehensive evaluation of our implementation, we developed a set of synthetic images, a small example of which is shown in Fig. 2. Here we have a 80×80 grid with 3×3 uniformly spaced, equally sized disks. The distance between the disks is equal to their radius. It is the disks that are to be segmented out using the maxflow algorithm. In our synthetic images the weights on the edges are as follows.

1. Source to center of disks (solid squares): 5×10^9 ,
2. Sink to non-disk regions (open circles): 5×10^9 ,
3. Between pixels wholly within disks or wholly outside disks: $10000 + r(100)$,
4. Extending between pixels within disks to pixels outside disks: $100 + r(10)$,

Here $r(n)$ is a random integer uniformly distributed between $-n$ and $+n$.

Fig. 2 is a small example to illustrate our approach—our actual measurements, reported below, are on grids of sizes 500^2 to 32000^2 . Problems with small disks (of radius 5-20 pixels) simulate pathology slides, while larger disks correspond to more general features.

While tests on synthetic images are no substitute for real image experiments, they do provide a standard framework for exploring the scaling of parallel algorithms. We show experiments on real images later in this paper.

3.4. Scheduling Global Relabeling

The normal way of using global relabeling is to apply the expensive breadth first search (BFS) relabeling after a fixed number of push relabel operations. The time required for the global relabeling step is dependent on the shape of the residual graph, which in turn depends on the pattern of flows in the original flow graph. This can make it difficult to choose one best fixed time interval for all types of graphs.

In our implementation, we have used a time factor approach. In this case, we measure the time required for each global relabel operation, t_{gr} , as the algorithm proceeds. The global relabel is applied only after the time spent on preflow-push operation t_{pp} exceeds ϕt_{gr} , where ϕ represents the *global relabeling factor*. The function $T_G()$ of Section 2.1.2 is therefore defined: $T_G(t_{pp}, t_{gr}, \phi): \text{if}(t_{pp} > \phi t_{gr}) \text{ true else false}$.

This is a more efficient way of scheduling the BFS, provided ϕ is chosen appropriately. Fig. 3 shows the performance of the parallel algorithm on a 4000^2 synthetic image with 200^2 disks on 40 processors of the Danriver XMT-2 machine. This plot shows that when ϕ is very large there is little benefit to using global relabeling, as would be expected. When ϕ is very small the global relabeling operation consumes most of the algorithm's time, leading to catastrophically poor performance.

There is a minimum that yields the best performance. In this plot the minimum occurs near $\phi = 3.0$. This is the value we have used in all of our remaining experiments. It is possible that $\phi = 3.0$ might not be the best choice for larger images and different disk sizes. Unfortunately, the empirical search for the best ϕ is a very time consuming operation and can only be justified in the case of a specific production code that is to be used repeatedly on a given class of problems.

A comparison of the performance of the traditional fixed interval approach versus our time factor approach (on 40 processors of the 64 processor Matterhorn XMT-2) is shown in Fig. 4. The plots in this figure show the active nodes in the flow graph against time and give an estimate of the progress of the algorithm. The problem being run here is the same as in Fig. 3, i.e. 4000×4000 image with 200×200 disks.

Global relabeling operations are indicated by solid rectangles, with the width of the rectangle representing the time required for that specific application of the relabeling operation (the heights of the rectangles are fixed and serve only to differentiate the rectangles from the underlying plots).

The plot labeled iN shows the progress of the algorithm for fixed interval N , i.e. N applications of preflow-relabel for every one global relabeling step. It is clear that infrequent operations lead to very poor performance. As the fixed interval is reduced from 1000 to 800 to 400, the performance improves. However, too small a time factor yields degradation in

performance because the algorithm spends most of its time doing expensive relabelings. This is evident from the crowded bars on the fixed $i30$ plot. In the case of the time factor approach, indicated by $\phi3.0$, the global relabeling operations are evenly spaced out with respect to the preflow-relabel operations and yield excellent performance.

4. PERFORMANCE ON SYNTHETIC IMAGES

The performance of our parallel preflow code is shown in Fig. 5, which includes comparisons with the highly optimized serial HiPR code.

In this case the image sizes vary from 500^2 to 32000^2 . In each case, for an n^2 image, the number of disks is $(n/20)^2$. In this figure the dashed line segments indicate the performance of the serial HiPR code. The dashed lines labeled 'S' show the performance of HiPR on one core of an 8 core Intel Sandybridge-EP CPU clocked at 2.00Ghz with 128GB of memory. Each core has its own 32kB L1 cache and 256kB L2 cache. The 8 cores on a CPU share a 20MB L3 cache. The lines labeled 'O' show the performance of HiPR on one core of a 2 core AMD Opteron 285 with 8GB memory running at 2.6GHz, with 1024 KB cache. For the smallest problems (500^2 & 1000^2) HiPR outperforms the parallel algorithm by a great extent. This is because the problem is too small for efficient parallelization on the XMT-2. In addition, the HiPR code contains many optimizations implementable only on serial machines that contribute to its performance. Finally, the caches on the the serial machines are also beneficial for small problems, as shown in Fig. 6 for the Intel Sandybridge-EP. Recall that the XMT has no cache—all memory locations are uniformly accessible.

The largest problem that HiPR could be run on the Opteron (with 8GB memory) was 3250^2 . The following discussion is in the context of the faster Sandybridge processor. For problems of size 2000^2 the parallel algorithm, at 8 processors, equals HiPR. The advantage of the parallel algorithm improves with increasing problem size. It is faster beyond 2 processors for problems of size 8000^2 . The remaining plots show good speedup, all the way to 32 processors for problems of size 2000^2 to 4000^2 and mostly up to 56 processors for the remaining. For all problems, there is a point where the performance starts deteriorating and it is not worthwhile to use more processors. This may be due to the problem being too small to afford parallelism for the given number of processors or because of limitations of the XMT-2's memory subsystem. The latter phenomenon has been described in [38], where it was shown that, for some XMT codes with intense memory access, it is worthwhile to use fewer than the available number of processors to get the best performance.

There is a spread in runtimes for large numbers of processors that we explored by running 5 randomly generated problems for sizes 500^2 to 8000^2 , for 8 processors or more. In these cases, the points indicate the individual runtimes, while the two diverging plots depict the envelope formed by the maximum and minimum run times. This spread could not be explored beyond 8000^2 because of the long run times required by the larger problems. For the same reason, entire ranges of processors could not be explored for the larger problem sizes.

The 500^2 to 16000^2 problems were run on the 64 processor Matterhorn machine, and these runs were terminated at 56 processors, as the performance had already started to deteriorate

at this point. The 24000^2 and 32000^2 problems were run on the 128 processor Danriver machine.

5. EXPERIMENTS WITH REAL IMAGES

We now describe the performance of our algorithm on a set of large, real images with different types of segmentation issues. These vary from features that extend across the entire extent of the image to features that are more localized.

The images that we analyze are

- 5.1 Confluence of the Amazon and Rio Negro rivers. This image is 2400×2400 pixels and contains very large features.
- 5.2 Spiral galaxy NGC 6872. This image is 5000×2812 pixels.
- 5.3 A pathology slide of size 5000×5000 pixels.
- 5.4 A digital photograph taken with the Hasselblad HD4-200MS 200 Megapixel camera. This image is 16352×12264 pixels.

The experiments described below are all on color images with Red-Green-Blue components specified in 1 byte each (and thus having values ranging from 0 to 255). Each pixel in the image corresponds to a node in the flow graph. The edge e_{ij} between two neighboring pixels is weighted using the euclidian RGB distance between the two pixels. For pixels i, j with RGB values $\langle R_i, G_i, B_i \rangle, \langle R_j, G_j, B_j \rangle$, respectively, this distance is

$$d_{ij} = \sqrt{(R_i - R_j)^2 + (G_i - G_j)^2 + (B_i - B_j)^2}.$$

The weight of the edge $w_{i,j}$ joining nodes i and j is

$$w_{ij} = Ae^{-d_{ij}/b},$$

where A and b are positive integers dependent on the image being segmented. Thus, w_{ij} is large when $d_{ij} = 0$ (similarly colored pixels) and is very small when $d_{ij} = \sqrt{3 \times 255^2}$ (pixels that are maximally different in color). This fulfills the requirements of edge weights for segmentation, as similar pixels tend to stay together and the cut tends to lie between dissimilar pixels.

We use $A = 10^4$ and $b = 2$ in the following. The choice of edge weights for segmentation flowgraphs is a research topic in itself. Our present purpose is to evaluate parallelization of the segmentation algorithm, and the simple formulation stated above suffices.

The source (sink) of the flow graph is connected to the grid of pixels with edges of ‘infinite’ weight, which in our case are 2^{60} . There are two modes for connecting the source/sink to the grid.

1. One or more discs, with specified radii, may be specified, and the source/sink connected to all nodes that lie within the discs.
2. One or more sets of RGB colors of interest may be specified and are used to derive RGB subcubes (i.e. the minimal cuboids of RGB space that contain the given set of colors). All pixels falling inside the subcubes are connected to the source/sink.

5.1. Confluence of the Amazon & Rio Negro

This experiment is on a 2400×2400 satellite image of the confluence of the Amazon and Rio Negro rivers near the city of Manaus in Brazil (Fig. 7). The waters of the Amazon flow down from the Andes mountains and contain heavy amounts of sediment (which shows as a muddy brown color). The Rio Negro, in contrast, flows through heavy jungle, where it collects vegetation that decomposes and imparts a dark color to the water. It is an interesting challenge for our segmentation code to differentiate between these waters, once they meet. In addition, it is also interesting to see how well the code can identify the numerous sub-channels of the two rivers as they flow through the jungle.

5.1.1. Creating the flow graph—To carry out this experiment for the Amazon, we sampled several areas of the muddy Amazon waters and used these to identify a Red-Green-Blue (RGB) color subcube with corners $\langle 113, 110, 90 \rangle$, $\langle 150, 133, 115 \rangle$. Pixels with colors within this subcube were connected to the source. Similarly, green areas of the jungle identified the subcube $\langle 47, 73, 63 \rangle$, $\langle 94, 109, 109 \rangle$ and pixels from this subcube were connected to the sink.

This experiment was repeated for the Rio Negro with a source subcube $\langle 36, 42, 61 \rangle$, $\langle 51, 57, 77 \rangle$, corresponding to the dark water, and an unchanged sink subcube.

5.1.2. Results—Fig. 8a shows the results of this experiment. The sediment-laden waters of the Amazon are clearly identified and the differentiation between the waters of the Amazon and Rio Negro is precise when they meet (top right). A smaller stream of the Amazon that whirlpools into the Rio Negro (middle left) is also identified. In addition, numerous subchannels of the Amazon flowing through the jungle are correctly delineated. Numerous small areas in the city of Manaus (top left) are also captured. These are caused by individual pixels that lie inside the color subcube and would, in practice, be eliminated using a filter.

Fig. 8b shows the results of the converse experiment, i.e. identifying the waters of the Rio Negro. The algorithm is equally successful in this case. However the cloud cover adjacent to the dark water obscures part of the channel. Some dark areas of vegetation near the city are incorrectly included.

In Fig. 8c we reveal the source pixels (green) used in Fig. 8a. Our objective here is to demonstrate that the source pixels are not sufficient to obtain a correct segmentation. The fine details of the segmentation boundaries are derived by the flow algorithm, using the source pixels as starting points. If the source pixels were enough, the entire regions bounded by red would be filled with green pixels, which is clearly not the case.

The traditional approach to using maxflow for image segmentation is to manually identify source and sink regions, typically using disks, as shown in Fig. 8d. This figure illustrates the fact that this approach leads to very poor segmentation, as compared with our automatic approach.

5.2. Spiral Galaxy

A composite 5000×2812 image of the largest known spiral galaxy NGC 6872 is shown in Fig. 9a. This image combines infrared and ultraviolet images. The ultraviolet areas indicate the presence of hot, young stars and are rendered in blue. We used the color subcube $\langle 12, 12, 31 \rangle$, $\langle 28, 51, 90 \rangle$, which corresponds to the blue haze, as source. The entire one pixel boundary of the image was used as sink.

Fig. 9b shows the resulting segmentation, which identifies the area of ultraviolet light surrounding the main galaxy NGC 6872. The main segmented area includes the smaller galaxy IC 4970 (vertical ellipse above main galaxy). Numerous other star systems that include blue areas are also identified. The most interesting area is the large blue patch on the top left, which is considered to be a new galaxy under development [43].

5.3. Pathology Slide

A segmentation experiment with a 5000×5000 pixel pathology slide is shown in Fig. 10. Although this experiment was run on the full slide, only 550×550 pixel extracts are shown in Figs. 10a and 10b.

Fig. 10a shows hematoxylin and eosin (H&E)-stained tissue taken from a prostate cancer biopsy (provided by The Cancer Genome Atlas <https://tcga-data.nci.nih.gov/tcga/>). With this staining, the nuclei of the cells (and a few other structures) are stained in hues of blue, eosinophilic structures (mostly intra- and extra-cellular proteins, including the cytoplasm) are stained in hues of pink and the red blood cells are stained in red. The output of the segmentation is shown in Fig. 10b, which results in nuclei segmentation. As can be observed from this figure, most of the relatively darker nuclei structures are successfully segmented while some of the background staining and cell clusters are not included in the segmentation results. It should be noted that nuclei segmentation is a challenging problem, and a series of pre- and post-processing steps are typically included to address problems such as stain non-uniformity, cell clustering and to achieve higher levels of accuracy. Further details on these steps and some solution strategies are discussed in detail in [44, 45].

The source and sink color subcubes in this case are $\langle 35, 11, 51 \rangle$, $\langle 71, 38, 100 \rangle$ and $\langle 81, 49, 105 \rangle$, $\langle 233, 182, 216 \rangle$, respectively.

5.4. Hasselblad

A digital photograph of a pair of shoes taken with the high resolution Hasselblad H4D-200MS digital camera is shown in Fig. 11. This 16352×12264 pixel image is one of the largest single digital camera images available on the web as of early 2013. The objective here is to show that very large, openly available, images can be processed successfully by our algorithm.

In this case we decided to segment out the light gray areas corresponding to the laces and the zig-zag decoration on the shoe. Accordingly, the RGB subcube $\langle 181, 181, 181 \rangle$, $\langle 255, 255, 255 \rangle$ was used for the source pixels. The grey pebble background was used as sink, with subcube $\langle 18, 18, 18 \rangle$, $\langle 101, 101, 101 \rangle$.

Three selected areas from the final segmented image are shown in Fig. 12. Fig. 12a shows that the laces and other light gray areas are correctly segmented. However, when the shade of gray falls outside the source subcube, the segmented area is cut short. Similarly, areas adjacent to the laces that also have the same gray colors, are included in the segmented regions. In Fig. 12b we further observe that some yellow regions adjacent to the grays are included because the RGB color distance between the yellows and grays is very small. A more sophisticated distance function would reduce this phenomenon. Some of the gray areas in Fig. 12c have specks in them. This is caused by imperfections in the shoe that result in tiny dark gray pixels (falling inside the sink subcube). Fig. 12c also illustrates the difficulty in segmenting highly textured areas (the purple regions and the region surrounding the “M&S” inscription). These textured areas have finely convoluted regions with high weight boundaries and result in dense, convoluted segmented areas. In addition, some small areas of the background pebbles are also included, since they happen to contain regions falling inside the source subcube. Filtering could be used to reduce these phenomena, but the topic is outside the scope of this paper.

5.5. Performance

Timings for the real image experiments described above are given in Fig. 13. The Amazon and Rio Negro experiments were run on the 64 processor Matterhorn while Slide ran on the 128 processor Danriver (each point in the plots is the average of 4 runs). Galaxy and Hasselblad were run on Danriver (each point represents one run). These results are generally consistent with the synthetic results of Fig. 5 demonstrating that the synthetic graphs are a good predictor of performance on real images.

It is interesting that the pathology slide, although larger than the Rio Negro image, has almost the same performance for 1 processor. Its performance improves more rapidly than Rio Negro and saturates slightly later than either Rio Negro or Amazon. This is because the segmented regions in the slide (Fig. 10b) are smaller, numerous and widely distributed. The Amazon and Rio Negro regions are large and extend across the entire image and are more stressful for the flow algorithm. This emphasizes the fact that the *type* of features being extracted influence the running time, in addition to the *size* of the image.

In the largest problem (Hasselblad, 16352×12264) performance saturates at about 64 processors. It is an interesting future challenge to refine our implementation of preflow-push to obtain even better scaling.

6. DISCUSSION

We have described the implementation and performance of the well-known preflow-push algorithm of Goldberg and Tarjan on the Cray XMT-2. This algorithm has met with great success in serial implementations but has been difficult to parallelize on conventional

parallel machines. We have shown that the algorithm can be implemented on the XMT-2 for grid graphs (of the type required for image segmentation) with very good performance. A number of large synthetic and real images were analyzed.

One of the constraining aspects of many parallel graph algorithms is the Breadth First Search (BFS) operation, which frequently limits the speedup of such algorithms. Several research groups have worked on efficient parallelization of BFS in both distributed memory [46] and in shared memory [47, 48, 49] parallel computers. Leiserson and Schardl [48], for example, propose a *bag* data structure for such algorithms. Our implementation uses the standard implementation of queues proposed by Gottlieb et al. [17] for BFS and obtains very good performance, which is a testament to the value of the fundamental operations available on the XMT-2. Of course, refinements of our code that use highly optimized BFS may lead to further improvements and are an interesting arena for future research.

The results of this paper demonstrate that the preflow-push algorithm can be implemented well on the XMT-2 and can be used to solve large and realistic problems in the image processing field, wherein maxflow graphs are grid-shaped.

The next steps in our research will be

1. Refinement of our implementation to further fine-tune performance using, among other things, better queuing primitives,
2. Extension of this work from 2D to 3D images (e.g. CAT scans and MRI images),
3. Extension of this work to maximum flow in general (i.e. non grid) graphs, which has numerous applications in fields like power grid stability, network optimization, etc., and
4. Evaluation of our implementation (and the extensions described above) on newer shared memory machines, such as those from Symmetric Computing, Convey and others.

Acknowledgments

We wish to thank Jeffrey Parvin for his encouragement of this research.

Access to the 16 processor ‘Egret’ XMT was provided by Cray Inc; we thank David Mizell and Kristyn Maschhoff at Cray for their assistance.

Access to the 128 processor ‘Cougar’ XMT was provided by the Center for Adaptive Supercomputing Software (CASS), Pacific Northwest National Laboratory (PNNL) (cass-mt.pnnl.gov). We thank John Feo, Richard Russell and Robert Adolf for their encouragement and Michael Peterson for help with the machine.

Access to the 64 processor ‘Matterhorn’ XMT-2 was provided by the Swiss National Supercomputing Centre (www.cscs.ch). We thank Thomas Schoenemeyer, Luca Marsella, and Neil Stringfellow for their generous assistance.

Access to the 128 processor ‘Danriver’ XMT-2 was provided by the Center for Applied High Performance Computing (CAHPC) at Noblis (www.noblis.org/MethodsTools/cahpc/Pages/solutions.aspx). We thank Benjamin Pecheux, William Mitchell, Philip Hayes and Curtis Bagby for their cooperation.

Shahid H. Bokhari was supported by Contract 181133 from the Pacific Northwest National Laboratory. Ümit V. Çatalyürek was supported, in part, by NSF grants CNS-0643969 and OCI-0904809.

The project described was supported in part by Award Number R01CA134451 from the National Cancer Institute to Metin N. Gurcan. The content is solely the responsibility of the authors and does not necessarily represent the official views of the National Cancer Institute, or the National Institutes of Health.

References

1. Gurcan M, Pan T, Sharma A, Kurc T, Oster S, Langella S, Hastings S, Siddiqui K, Siegel E, Saltz J. Gridimage: A novel use of grid computing to support interactive human and computer-assisted detection decision support. *Journal of Digital Imaging*. 2007; 20:160–171. 10.1007/s10278-007-9020-0. URL <http://dx.doi.org/10.1007/s10278-007-9020-0>. [PubMed: 17318701]
2. Gurcan M, Boucheron L, Can A, Madabhushi A, Rajpoot N, Yener B. Histopathological image analysis: A review. *Biomedical Engineering, IEEE Reviews in*. 2009; 2:147–171. 10.1109/RBME.2009.2034865.
3. Cooper L, Sertel O, Kong J, Lozanski G, Huang K, Gurcan M. Feature-based registration of histopathology images with different stains: an application for computerized follicular lymphoma prognosis. *Comput Methods Programs Biomed*. Dec; 2009 96(3):182–192. [PubMed: 19487043]
4. Swanson MS, Prescott JW, Best TM, Powell K, Jackson RD, Haq F, Gurcan MN. Semi-automated segmentation to assess the lateral meniscus in normal and osteoarthritic knees. *Osteoarthr Cartil*. Mar; 2010 18(3):344–353. [PubMed: 19857510]
5. Sertel, O.; Kong, J.; Lozanski, G.; Shanaah, A.; Çatalyürek, ÜV.; Saltz, JH.; Gurcan, MN. ICASSP. IEEE; 2008. Texture classification using nonlinear color quantization: Application to histopathological image analysis; p. 597-600.
6. Ababneh SY, Prescott JW, Gurcan MN. Automatic graph-cut based segmentation of bones from knee magnetic resonance images for osteoarthritis research. *Medical Image Analysis*. Aug; 2011 15(4):438–448. 10.1016/j.media.2011.01.007. [PubMed: 21474362]
7. Sertel O, Kong J, Catalyurek UV, Lozanski G, Saltz JH, Gurcan MN. Histopathological Image Analysis Using Model-Based Intermediate Representations and Color Texture: Follicular Lymphoma Grading. *Journal of Signal Processing Systems for Signal Image and Video Technology*. Apr; 2009 55(1-3):169–183. 10.1007/s11265-008-0201-y.
8. Cray Inc. Cray XMT System Overview. 2008 Document S-2466-12.
9. Alverson G, Alverson R, Callahan D, Koblenz B, Porterfield A, Smith B. Exploiting heterogeneous parallelism on a multithreaded multiprocessor. *Proc Int Conf Supercomputing*. 1992:188–187.
10. Alverson R, Callahan D, Cummings D, Koblenz B, Porterfield A, Smith B. The Tera computer system. *Proc Int Conf Supercomputing*. 1990:1–6.
11. Greig D, Porteous B, Seheult A. Exact maximum A Posteriori estimation for binary images. *Journal of the Royal Statistical Society. Series B*. 1989; 51(2):271–279.
12. Boykov Y, Funka-Lea G. Graph cuts and efficient N-D image segmentation. *International Journal of Computer Vision*. Nov; 2006 70(2):109–131. 10.1007/s11263-006-7934-5.
13. Delong, A.; Boykov, Y. A scalable graph-cut algorithm for N-D grids. 2008 IEEE Conference on Computer Vision and Pattern Recognition; Anchorage, AK 345 E 47th St, New York, NY 10017 USA: IEEE; 2008. p. 946-953.
14. Boykov Y, Kolmogorov V. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Sep; 2004 26(9):1124–1137. 10.1109/TPAMI.2004.60. [PubMed: 15742889]
15. Jamriska, O.; Sykora, D.; Hornung, A. Cache-efficient graph cuts on structured grids. *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference*; 2012. p. 3673-3680. 10.1109/CVPR.2012.6248113
16. Goldberg A, Tarjan R. A New Approach to the Maximum-Flow Problem. *Journal of the ACM*. Oct; 1988 35(4):921–940. 10.1145/48014.61051.
17. Gottlieb A, Grishman R, Kruskal CP, McAuliffe KP, Rudolph L, Snir M. The NYU Ultracomputer: Designing an MIMD shared memory parallel computer. *IEEE Trans Comput*. 1983; 32(2):175–189. <http://dx.doi.org/10.1109/TC.1983.1676201>.
18. Ford L, Fulkerson D. Maximum flow through a network. *Canadian Journal of Mathematics*. 1956; 8:399–404.

19. Edmonds J, Karp RM. Theoretical improvements in algorithmic efficiency for network flow problems. *J ACM*. 1972; 19(2):248–264.
20. Dinitz E. Algorithm for solution of a problem of maximum flow in networks with power estimation. *Soviet Math Dokl*. 1970; 11:1277–1280.
21. Karzanov AV. Determining the maximum flow in a network by the method of preflows. *Soviet Mathematics Doklady*. 1974; 15:434–437.
22. Goldberg, AV.; Tardos, E.; Tarjan, RE. Network flow algorithms. In: Korte, BH.; Lovasz, L.; Proemel, HJ.; Schrijver, A., editors. *Paths, Flows, and VLSI-Layout. Algorithms and Combinatorics*, Springer-Verlag; 1990. p. 101-164.
23. Goldberg, AV.; Hed, S.; Kaplan, H.; Tarjan, RE.; Werneck, RF. Maximum flows by incremental breadth-first search. *Proceedings of the 19th European conference on Algorithms, ESA'11*; Berlin, Heidelberg: Springer-Verlag; 2011. p. 457-468. URL <http://dl.acm.org/citation.cfm?id=2040572.2040623>
24. Cherkassky BV, Goldberg AV. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*. 1997; 19(4):390–410.
25. Derigs U, Meier W. Implementing Goldberg's max-flow-algorithm-a computational investigation. *Methods and Models of Operational Research*. 1989; 33:383–403.
26. Alizadeh, F.; Goldberg, A. Technical Report STAN-CS-92-1410. Computer Science Department; Stanford University; Feb. 1992 On implementing the push-relabel method for the maximum flow problem on a Connection Machine.
27. Anderson R, Setubal J. A Parallel Implementation of the Push-Relabel Algorithm for the Maximum Flow Problem. *Journal of Parallel and Distributed Computing*. Aug 15; 1995 29(1):17–26. 10.1006/jpdc.1995.1103.
28. Bader DA, Sachdeva V. A cache-aware parallel implementation of the push-relabel network flow algorithm and experimental evaluation of the gap relabeling heuristic. *ISCA Int Conf on Parallel and Dist Comp Sys*. 2005:41–48.
29. Shekhovtsov A, Hlav V. A distributed mincut/maxflow algorithm combining path augmentation and push-relabel. *International Journal of Computer Vision*. 2013; 104(3):315–342. 10.1007/s11263-012-0571-2. URL <http://dx.doi.org/10.1007/s11263-012-0571-2>.
30. Hong B, He Z. An asynchronous multithreaded algorithm for the maximum network flow problem with nonblocking global relabeling heuristic. *Parallel and Distributed Systems, IEEE Transactions on*. 2011; 22(6):1025–1033. 10.1109/TPDS.2010.156.
31. Snaveley, A.; Carter, L.; Boisseau, J.; Majumdar, A.; Gatlin, KS.; Mitchell, N.; Feo, J.; Koblenz, B. Multi-processor performance on the Tera MTA. *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*; 1998. p. 1-8.
32. Anderson, W.; Briggs, P.; Hellberg, CS.; Hess, DW.; Khokhlov, A.; Lanzagorta, M.; Rosenberg, R. Early experience with scientific programs on the Cray MTA-2. *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*; Washington, DC: IEEE Computer Society; 2003.
33. Feo, J.; Harper, D.; Kahan, S.; Eldorado, Konecny P. *CF '05: Proceedings of the 2nd International Conference on Computing Frontiers*; New York, NY: ACM; 2005. p. 28-34. <http://doi.acm.org/10.1145/1062261.1062268>
34. Mizell, D.; Maschhoff, K. Early experiences with large-scale Cray XMT systems. *Parallel and Distributed Processing Symposium, International*; 2009. p. 1-9. <http://doi.ieeeecomputersociety.org/10.1109/IPDPS.2009.5161108>
35. Bokhari SH, Sauer JR. Sequence alignment on the Cray MTA-2. *Concurrency and Computation*. 2004; 16:823–839.
36. Bokhari, S.; Sauer, J. Parallel algorithms for bioinformatics. In: Zomaya, A., editor. *Parallel Computing for Bioinformatics*. Wiley; 2006. p. 509-529.
37. Bokhari S, Saltz J. Exploring the performance of massively multithreaded architectures. *Concurrency and Computation: Practice and Experience*. 2010; 22(5):588–616.
38. Bokhari SH, Bokhari SS. A comparison of the Cray XMT and XMT-2. *Concurrency and Computation: Practice and Experience*. 2013; 25(15):2123–2139. 10.1002/cpe.2909. URL <http://dx.doi.org/10.1002/cpe.2909>.
39. Cray Inc. *Cray XMT Performance Tools User's Guide*. 2011:S-2462–20.

40. Bokhari SH, Glaser MA, Jordan HF, Lansac Y, Sauer JR, Van Zeghbroeck B. Parallelizing a DNA simulation code for the Cray MTA-2. *Proc IEEE Computer Soc Bioinformatics Conf.* 2002;291–302.
41. Bokhari SH, Sauer JR. A parallel graph decomposition algorithm for DNA sequencing with nanopores. *Bioinformatics.* 2005; 21(7):889–896. [PubMed: 15539451]
42. Bokhari SS. Parallel solution of the subset-sum problem: an empirical study. *Concurrency and Computation: Practice and Experience.* 2012; 24(18):2241–2254. 10.1002/cpe.2800. URL <http://dx.doi.org/10.1002/cpe.2800>.
43. Eufrazio, RT.; De Mello, DF.; Urrutia-Viscarra, F.; Mendes de Oliveira, C.; Dwek, E. When the UV Unveils the Largest Spiral. *American Astronomical Society Meeting Abstracts; American Astronomical Society Meeting Abstracts*; 2013. #147.10. URL <http://adsabs.harvard.edu/abs/2013AAS...22114710E>
44. Kong H, Gurcan M, Belkacem-Boussaid K. Partitioning histopathological images: An integrated framework for supervised color-texture segmentation and cell splitting. *Medical Imaging, IEEE Transactions on.* Sep; 2011 30(9):1661–1677. 10.1109/TMI.2011.2141674.
45. Kong H, K BB, Gurcan M. Cell nuclei segmentation for histopathological image analysis. *SPIE Medical Imaging.* 2011; 7962
46. Yoo, A.; Chow, E.; Henderson, K.; McLendon, W.; Hendrickson, B.; Çatalyürek, ÜV. A scalable distributed parallel breadth-first search algorithm on BlueGene/L. *Proceedings of SC2005 High Performance Computing, Networking, and Storage Conference*; 2005.
47. Bader DA, Madduri K. SNAP, small-world network analysis and partitioning: An open-source parallel graph framework for the exploration of large-scale networks. *International Symposium on Parallel and Distributed Processing (IPDPS).* 2008:1–12.
48. Leiserson CL, Schardl TB. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). *Symposium on Parallel Architectures and Algorithms (SPAA).* 2010:303–314.
49. Saule, E.; Çatalyürek, ÜV. An early evaluation of the scalability of graph algorithms on the Intel MIC architecture. *26th International Symposium on Parallel and Distributed Processing, Workshops and PhD Forum (IPDPSW), Workshop on Multithreaded Architectures and Applications (MTAAP)*; 2012.

Figure 1.

Transforming a segmentation problem into a maxflow problem. In this illustrative example, the source and sinks are connected to a few nodes in the grid. In actual practice they are connected to collections of nodes corresponding to regions on the image that are unambiguously associated with either the source or the sink. The arrows show a representative flow in the graph—the underlying edges may be undirected or bidirected.

Figure 2.

An example synthetic image. Here we have an 80×80 grid with 3×3 disks, each with radius 7. Each symbol represents a pixel. Pixels marked with solid squares are connected to the source, while those marked with open circles are connected to the sink. Actual synthetic images used in our experiments are much larger, ranging from 500^2 to 32000^2 pixels.

Figure 3.

Time taken by our implementation when segmenting a 4000^2 image with 200^2 disks on 40 processors of the 128 processor Danriver XMT-2. The global relabeling heuristic, which involves breadth first search, is applied whenever the cumulative time for push-relabels exceeds ϕ times the last measured global relabeling time. ϕ is the *global relabeling factor*. Four randomly generated problems are shown with different colors.

Figure 4.

A comparison of the the parallel preflow-push algorithm on the same problem as in Fig. 3 on 40 processors of the 64 processor Matterhorn XMT-2. The performance of the fixed interval approach is shown by the plots labeled iN , where N is the number of preflow-relabels between applications of global relabel. The time factor approach is shown by $\phi 3.0$. Solid rectangles represent global relabeling steps.

Figure 5.

Performance of our parallel code for problems of size 500^2 to 32000^2 . A problem of size n^2 has $(n/20)^2$ disks. The 500^2 to 16000^2 problems were run on the 64 processor Matterhorn machine, while 24000^2 and 32000^2 were run on the 128 processor Danriver machine. For problems of size 500^2 to 8000^2 , we ran 5 randomly generated problems for 8 or more processors to illustrate the spread of runtimes. This could not be done for problems larger than 8000^2 because of the large amounts of time required for each run. As a result the plot for 16000^2 shows a significant deterioration at 56 processors. Multiple runs at 56 processors would have shown a spread of timings. Only limited ranges of processors could be explored for problems larger than 8000^2 for the same reason. The horizontal dashed lines show the performance of Goldberg's serial HiPR code running on two different serial processors.

Figure 6.

Impact of cache when running HiPR on a 2.0GHz, 128GB Intel Sandybridge-EP. A significant portion of HiPR's data structures fit in this machine's caches for problems less than 1400^2 , leading to excellent performance for very small problem sizes as indicated by a clear discontinuity.

Figure 7.

In this 2400×2400 pixel image, taken from NASA's Earth Observatory (<http://earthobservatory.nasa.gov/NaturalHazards/view.php?id=39359>), the Amazon and Rio Negro rivers meet near the city of Manaus in Brazil. The Amazon is rich in sediment and is colored brown. The Rio Negro contains decomposed vegetation and shows up as black. The two rivers flow side by side after the confluence (top right), posing an interesting challenge for our segmentation algorithm.

Figure 8.

Segmenting the confluence of the Amazon and Rio-Negro.

- (a) The algorithm clearly segments out the waters of the Amazon, as well as several narrow substreams. The distinction between the waters of the two rivers after confluence (top right) is precisely identified, despite some cloud cover.
- (b) Segmenting the waters of the Rio Negro. The main river is correctly identified, as are numerous small substreams. Dark areas of vegetation, including some near the city are incorrectly identified. These would be eliminated by a more elaborate distance function.
- (c) Bright green pixels show the source pixels used in Fig. 8a. These pixels are not enough for correct segmentation: the finer details are identified by the flow algorithm.
- (d) Using a traditional source-sink based segmentation approach yields poor results (compared with Fig. 8a). The green (blue) disks are manually placed sources (sinks).

(a)

(b)

Figure 9.

The spiral galaxy NGC 6872 is shown in this composite 5000×2812 pixel image, taken from the Jet Propulsion Laboratory (<http://www.jpl.nasa.gov/spaceimages/details.php?id=PIA16613>).

(a) Ultraviolet light is rendered as a bluish haze and indicates the presence of hot, young stars.

(b) The segmentation algorithm delineates the regions of ultraviolet emission and clearly identifies a nascent dwarf galaxy (top left).

Figure 10.

A 550×550 pixel extract from a 5000×5000 pathology slide.

Figure 11.

A 16352×12264 digital image captured with a Hasselblad HD4-200MS camera by Anders Espersen (http://www.hasselblad.com/media/3722362/salomon_h4d-200ms.zip). Our experiment is to segment the light gray laces and zig-zag decoration. Used with permission.

Figure 12. Selected areas from the segmented Hasselblad image. For clarity, different colors are used to indicate boundaries.

Figure 13.
Performance of our parallel code for real images.