A Multi-Level Parallel Implementation of a Program for Finding Frequent Patterns in a Large Sparse Graph

Steve Reinhardt[1] and George Karypis[2]

[1]SGI, 2750 Blue Water Road, Eagan, MN 55121 USA spr@sgi.com
[2]Department of Computer Science and Engineering, University of Minnesota, Minneapolis, MN 55455 USA karypis@cs.umn.edu

Abstract

Graphs capture the essential elements of many problems broadly defined as searching or categorizing. With the rapid increase of data volumes from sensors, many application disciplines need to process larger graphs quickly. This paper presents the results of parallelizing with OpenMP an algorithm that finds, in a single large labeled undirected sparse graph, the connected subgraphs with a given minimum number of edge-disjoint embeddings. Parallelism is exploited at two levels in the algorithm. The lack of a priori knowledge of the extent of parallelism for a given input required use of a dynamic, multi-level approach based on the proposed OpenMP `taskq/task` *extensions. The parallel implementation required the addition of 21 directives and about 50 accompanying lines of code, in an original code of about 15,000 lines. Experimental results show excellent speed-up to 30 processors for the graphs used, with a best speed-up of 26.1 compared to the serial version. The* `taskq/task` *constructs show promise for problems exhibiting unstructured parallelism.*

Keywords: pattern discovery, frequent subgraph, data mining, parallel processing, OpenMP, unstructured parallelism

1 Introduction

The use of graphs to represent discrete data and the connections within the data is growing as researchers seek to solve a variety of problems in data mining, chemical compound classification, link analysis, and computational biology. The size of these graphs is growing explosively, corresponding to the growth of the data volume from sensors (*e.g.,* microscopes, mass spectrometers, and gene arrays). Computational methods for these large graphs need to be fast for routine use, and thus one is naturally drawn to parallel extension of the existing serial methods.

This work builds on Kuramochi and Karypis' VSiGraM algorithm [1], extending it to exploit parallelism both across candidate subgraphs, and within a candidate subgraph on the multiple embeddings in the large graph. This approach allows exploitation of a large amount of the potential parallelism within the algorithm. However, the two levels of parallelism and the recursive nature of the algorithm do not map trivially to the existing parallel language methods. After two aborted attempts with other parallel methods, we finally used the `taskq` and `task` extensions [5] to the OpenMP 2.5 [7] parallel interface. These constructs support parallelism at multiple locations or levels within a program, with the run-time library directing processors to a single conceptual work queue. The desired parallelism within the algorithm mapped naturally to these constructs. We implemented this algorithm so that the parallelism between subgraphs (*high level*) and within a subgraph (*low level*) could be turned on or off independently, to measure the performance benefit of each. The high-level parallelism consistently showed strong benefit, but the low-level parallelism showed only modest, and often no, benefit for the graphs we used, and indeed degraded performance as more processors were added. Our experience with this algorithm and OpenMP showed that the `taskq/task` constructs are a promising means of effectively parallelizing this type of algorithm, both in terms of the time spent changing an existing serial code and in the performance benefit of parallelism.

2 Problem Statement

The problem considered here, as described in [1], is that of a graph G consisting of a set of vertices V and a set of edges E, with each vertex and edge having a (potentially non-unique) label. A graph G_s is a *subgraph* of G if and only if $V_s \subseteq V$ and $E_s \subseteq E$. Two graphs G_1 and G_2 are isomorphic if they are topologically identical to each other. The problem of **subgraph isomorphism** between G_1 and G_2 is to find whether an isomorphism between G_2 and and a subgraph of G_1. Two embeddings of G_s are identical if they use the same set of edges of G_s, and they are called edge-disjoint if they do not have any edges of G in common. This work focuses on finding subgraphs of a single large graph whose embeddings are edge-disjoint.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

The overlap graph of G_s is a graph obtained by creating a vertex for each non-identical embedding, in the set of all embeddings of G_s in G and allocating an edge for each pair of embeddings that are not edge-disjoint. Calculation of the set of edge-disjoint embeddings depends on calculating the maximal independent set of the overlap graph. One input parameter to the algorithm is the minimum **frequency** of occurrence of a subgraph to consider. Because of the potential computational complexity of the exact calculation of MIS, the vSiGraM program embodies three different algorithms for calculating MIS: one heuristic and one exact algorithm for calculating the MIS and one exact algorithm for calculating an upper-bound on the MIS.

3 Approach to Parallelization

The original SiGraM work had different implementations for a depth-first or *vertical* version and a breadth-first or *horizontal* version. With the inclusion of parallelism, the notion of "first" becomes imprecise, as does the distinction between depth-first and breadth-first, as the parallelism will typically be proceeding in both dimensions simultaneously. This work was based on the vertical version of VSiGraM, as the recursive structure of that code expressed the potential parallelism well.

Conceptually, the algorithm has several dimensions of parallelism that could be exploited. The algorithm (with slight modifications from Algorithm 5 from [1]) is shown in Figure 1. vSiGraM creates the size-1 subgraphs (each represented by F^1) and then calls vSiGram-Extend with a subgraph, the whole graph, and the minimal frequency f. vSiGraM-Extend calls itself recursively, and hence most of the time is spent there. In vSiGraM-Extend, lines 2-3 creates all the size-*i+1* subgraphs ($\mathbf{C}^{i+1}$) that contain F^i, then prunes the set for redundancy (choosing only those whose generating parent is the candidate subgraph) and minimal frequency. All of those surviving subgraphs are then extended again recursively.

Figure 1. Algorithm 5 from Kuramochi and Karypis [1].

```
vSiGraM (G, MIS_type, f)
1.  F ← ∅
2.  F^1 ← all frequent size-1 subgraphs in G
3.  for each F^1 in F^1 do
4.        M(F^1) ← all embeddings of F^1
5.  for each F^1 in F^1 do
6.        F ← F ∪ vSiGraM-Extend(F^1, G, f)
7.  return F

vSiGraM-Extend(F^k, G , f)
1.  F ← ∅
2.  for each embedding m in M(F^k) do
3.        C^k+1 ← C^k+1 ∪ {all (k+1)-subgraphs of G containing m}
4.  for each C^k+1 in C^k+1 do
5.        if F^k is not the generating parent of C^k+1 then
6.              continue
7.        compute C^k+1.freq from M(C^k+1)
8.        if C^k+1.freq < f then
9.              continue
10.       F ← F ∪ vSiGraM-Extend(C^k+1, G, f)
11.return F
```

The high-level parallelism exploited is at the parallel loop of lines 5 and 6 of vSiGraM and the recursion step in line 10 in vSiGraM-Extend. At these points, each of the candidate size-*i* subgraphs represents a distinct set of vertices and edges, and its suitability for extension to size *i+1* can be determined independently of the suitability of any other size-*i* subgraph. *I.e.*, it depends only on the input graph and the candidate subgraph itself, and in no way on the characteristics of other subgraphs. Thus each of the size-*i* subgraphs can be extended in parallel. This parallelism grows directly with the number of candidate subgraphs.

The low-level parallelism exploited is lines 2 and 3 of vSiGraM-Extend. These lines find all the possible size-*i+1* subgraphs that contain the size-*i* subgraph, storing the unique subgraphs that result with the count of occurrences. This parallelism tends to grow as the size of the graph grows.

3.1 Possible Parallel Languages

The dominant parallel programming method in use today is MPI [2], but the characteristics of this problem are a poor match for it. In general for large

problems, each processor will want to traverse the original large graph as needed, and that graph will be big enough not to fit in the memory of a single node (processor plus memory) of the parallel system. Ideally the code could address directly into the large array, with the array being distributed across the memory of multiple nodes. Also, the extent of parallelism is dynamic and typically highly imbalanced, necessitating a means of balancing the work across the processors. While load-balancing abilities have been implemented for MPI in some packages (e.g., Zoltan [3]), they are not simple and straight-forward to implement in a modestly sized (~15,000 lines of C) research code. One of the criteria for this work was that the changes for parallelism be minimally intrusive to the structure of the existing code.

We tried to use Unified Parallel C (UPC) [4], a global-address-space extension to the C language intended for problems with fine-grained, dynamic parallelism. After considerable time working to map the existing vSiGraM code onto the UPC constructs, we abandoned that approach. The primary difficulties were in defining structures properly so that they could be allocated and addressed appropriately (private or shared) in different circumstances, in the context of an existing code that has many different structures defined with pointers to each other. We wound up with myriad combinations, for instance with a flavor of struct A containing a pointer to a private struct B and a distinct flavor of struct A containing a pointer to a shared struct B. Researchers more experienced with UPC may have succeeded where we did not.

Next we used OpenMP. OpenMP is rarely used in today's HPC clusters, in part because until recently the nodes in those clusters had just one or two processors in a cache-coherent domain (a near-requirement for OpenMP implementation), and thus there was little opportunity for performance benefit. With the onset of multi-core processors, however, several manufacturers already offer inexpensive nodes with 4-8 cores, and one can reasonably expect those nodes to have 10s of cores in the near future. This makes OpenMP a reasonable choice for parallelism on the node. A few manufacturers (SGI most extremely) offer large cache-coherent nodes that could be used for problems that don't fit on the typical small node.

Our first OpenMP implementation exploited just high-level parallelism, using the parallel `for` construct. We used the Electric Fence [6] library to find data erroneously typed as `private` or `shared` and quickly had a working code. However, this code did not exhibit the expected level of speed-up, due to the dynamic nature of the parallelism and the static nature of allocation of processors to standard OpenMP parallel regions. At the time of creation of a parallel region, the programmer must specify how many threads (processors) are to be used in the region, and those threads are dedicated to that region for the duration of the region. In the recursive nature of vSiGraM, this meant that threads were allocated to the parallel region at depth *i*, and were unavailable to participate in the parallel regions at other depths. A more dynamic means of allocation of threads and tasks was needed. Also, we wanted to exploit the low-level parallelism, which was dynamic and also orthogonal to the high-level parallelism.

The proposed `taskq/task` extensions to OpenMP have been implemented in the latest Intel C/C++ compiler [5]. They allow disparate locations in a program to add work to a single conceptual queue that is served by all the threads active in the program. Multiple queues can also be created, with the run-time library scheduling threads dynamically among the queues. This matches very well with the desired parallelism (recursive, multi-location) in vSiGraM. A simple single-queue example of the use of `taskq/task` for a recursive subroutine is as follows:

```
main()
{
   int val;
#pragma intel omp taskq
   val = fib(12345);
}

fib(int n)
{
   int ret[2];
   if (n>2)
#pragma intel omp task
      for(i=n-2; i<n; i++) {
         ret[n-2-i] = fib(i);
      }
      return (ret[0] + ret[1]);
   } else {
      return 1;
   }
}
```

The first pragma declares the task queue and the second pragma designates items to be placed on the queue and hence executed. The `taskq` pragma is placed outside the recursive routine so that all the items will be added to a single queue.

The second vSiGraM OpenMP implementation was based on `taskq/task`, with switches to allow the high-level and low-level parallelism to be enabled independently for measurement purposes. Implementation of the high-level parallelism was straight-forward, with a parallel region around the extension of the original size-1 subgraphs and a parallel construct around the recursive call to create size-*i+1* subgraphs. Key portions of the parallel code are presented in Figure 2.

Implementation of the low-level parallelism required more thought. This code expands the map of the current subgraph to identify all valid size-*i+1* extensions. Those extensions could occur more than once, which the algorithm checks for before registering an extension in the extension set. When run serially, this check is simple. The code is

```
for (i=0; i < vmap_sz(sg); i++)
  expand_map(sg, ct, ams, i,es,lg);
```

When run in parallel, however, two processors may identify the new extension nearly simultaneously. These duplicate extensions which must be combined in the result extension set. Thus after the parallel loop finds all the locally distinct extensions, a reduction step collapses them to all the globally distinct extensions. Because of the semantics of the `taskq/task` constructs, where non-master threads are not guaranteed to execute the postamble of the `task` loop, an extra data structur, suggested by Grant Haab and colleagues from the Intel OpenMP development group, is used to store the local extension sets for all the non-master threads, and then the master does the reduction in the postamble. The code for the parallel version is in Figure 3. This implementation of high- and low-level parallelism produced identical results as the serial algorithm for the number of frequent subgraphs of each size, the number of maps, the number of live edges, etc.

The changes to use OpenMP included 12 OpenMP directives in the main algorithms. (One of these directives and accompanying code changes was necessary to enable running with high- and low-level parallelism independently, which would probably not persist into a final implementation.) Nine OpenMP directives were necessary in utility routines (memory allocation, mainly) to ensure proper `private/shared` definition and mutual exclusion.

Figure 2. Parallel Code for "High Level" Parallel Section

```
 // In a top-level routine, start all the initial subgraphs
#pragma intel omp parallel taskq shared(i,freq)  default(none)
                for (i = 0; i < sg_set_size(freq); i++) {
#pragma intel omp task captureprivate(i)
                        {
                        SubGraph *sg = sg_set_at(freq, i);
                        expand_subgraph(sg, ct, lg, ls, o);
                        }
                }
        }
```

```
//  At the bottom of expand_subgraph, after all child subgraphs
//  have been identified, start them all.
#pragma intel omp taskq
        for (ii=0; ii<sg_set_size(child); ii++) {
#pragma intel omp task captureprivate(ii)
                {
                SubGraph *csg = sg_set_at(child,ii);
                expand_subgraph(csg, csg->ct, lg, ls, o);
                }       // end-task
        }
```

One difficulty we did not resolve, but merely deferred, was that of expanding an array after it had been once allocated. In a serial version, this is simple, as one can know which structures point to a given array and go update those structures to point to a newly allocated array containing the contents of the formerly allocated array. Updating these pointers is, in general, not possible in a parallel situation. We deferred this problem by making arrays substantially larger than they would have normally been, so we would rarely hit the array-overflow conditions, but we still hit them in some of the larger runs with more processors.

Figure 3. Parallel Code for "Low Level" Parallel Section (some details omitted)

```
#pragma omp parallel shared(nt, priv_es)
    {
#pragma omp master
        {
                nt = omp_get_num_threads();   //#threads in par
                priv_es = (ExtensionSet **)kmp_calloc(nt, sizeof(ExtensionSet *));
        }
#pragma omp barrier
#pragma intel omp taskq
        {
                for (i = 0; i < sg_vmap_size(sg); i++) {
#pragma intel omp task captureprivate(i)
                        {
                                int th = omp_get_thread_num();
                                if (priv_es[th] == NULL) {
                                        priv_es[th] = exset_init(128);
                                }
                                expand_map(sg, ct, ams, i, priv_es[th], lg);
                        }
                }
        }
     }
        for (i=0; i < nt; i++) {
                if (priv_es[i] != NULL) {
                        exset_merge(priv_es[i],es);
                }
        }
        kmp_free(priv_es);
```

Figure 4. VSiGraM Timings and Scaling for DTP Input Data

Note: Serial times use neither high-level nor low-level parallelism.

Graph	Frequency	Type of Parallelism	Number of processors						
			1	2	4	8	16	30	60
			Time in seconds (speed-up)						
dtp	500	High	31.94	17.01 (2.03)	14.76 (2.40)	13.89 (2.58)	14.00 (2.56)	13.97 (2.57)	
		Low		32.51 (0.98)	31.52 (1.01)	37.95 (0.83)	42.18 (0.74)	49.56 (0.63)	
		Both		17.52 (1.96)	14.88 (2.37)	15.80 (2.21)	29.85 (1.08)	44.37 (0.70)	
	100	High	93.96	48.86 (1.97)	27.12 (3.71)	16.82 (6.39)	15.05 (7.29)	14.52 (7.61)	
		Low		94.36 (1.00)	92.18 (1.02)	112.17 (0.83)	133.40 (0.70)	116.31 (0.80)	
		Both		48.38 (1.99)	27.27 (3.69)	61.52 (1.55)	315.94 (0.29)	281.83 (0.33)	
	50	High	282.15	142.02 (2.00)	62.73 (4.64)	34.44 (8.76)	19.40 (16.56)	15.06 (22.27)	15.80 (21.03)
		Low		283.19 (1.00)	293.6 (0.96)	400.55 (0.70)	262.82 (1.07)	197.27 (1.44)	
		Both		140.47 (2.03)	81.18 (3.55)	242.09 (1.17)	513.39 (0.55)	581.04 (0.48)	

4 Experimental Results

The code was compiled with the Intel C++ compiler, version 9.0.030, with the `-openmp_profile` and `-O3` flags. The results below were obtained on an SGI Altix 4700 system populated with 32 1.6GHz Itanium2 dual-core "Montecito" sockets (64 cores) and 64GB of memory. No special `dplace` or `cpuset` commands were used. Attention was paid to scaling performance, and not to absolute per-processor performance. The runs all used the upper-bound algorithm [1] for maximum independent set (MIS) calculation. The times are all in seconds of wall-clock time, selected as the minimum of 4 runs except the serial runs, which are the minimum of 2 runs. The minimums were chosen because the programs were run on quiet, but not strictly dedicated systems, so system effects could have degraded results. In practice, the times of the multiple runs were similar. The processing of the input graph takes about 2.5 seconds, in a serial region. The speed-up calculation excludes the time spent reading the input file.

We used two of the graphs explored in [1] for this work. The first graph was the DTP graph, which represents chemical compounds (vertices being atoms, edges being bonds). The connected components are of modest size (average 21), with many similar chemical idioms being likely. The Air1 graph represents aviation safety information and has many more found patterns (see Table 3 in [1]) than DTP.

Figure 5. Sample Graph Data

	#components	min	max	avg
	#vertices	#edges	#vertex labels	#edge labels
Air1	2,606	21	55	38
	101,088	98,482	6,171	51
DTP	2,080	1	110	21
	40,879	43,070	52	3

4.1 Results with DTP graph

We obtained a comprehensive set of results with the DTP graph. High-level parallelism showed good speed-ups for the frequency settings where there was sufficient work to employ the number of processors. For the highest tested frequency of 50, the best speed-up was 22.27 on 30 processors. However, even for that frequency, the marginal benefit of parallelism was greatly reduced when going from 16 to 30P, as the parallel efficiency dropped from 1.04 to 0.74. The only experiment with a greater processor count (60P) showed worse absolute performance. Only in one case did low-level parallelism show any benefit outside of measurement error, and typically it resulted in significantly worse absolute performance. Because of the poor performance of low-level parallelism, the combination of high- and low-level parallelism was also poor.

4.2 Results with Air1 Graph

For the Air1 graph, we focused on high-level parallelism, since that's what had shown benefit for the DTP graph, and on lower frequency thresholds and larger processor counts (up to 60P for all frequencies), to push the limits of scaling. The results were very similar to those for the DTP graph; namely that scaling of the high-level algorithm was very good to 8P but not as good at 30P, and the marginal benefit of going to 60P was modest and in some cases negative. The only trial with low-level parallelism showed the best benefit from it for any case, with an absolute speed-up of 2.13 on 60P.

Figure 6. VSiGraM Timings and Scaling for Air1 Input Data
Note: Serial times use neither high-level nor low-level parallelism.

Graph	Frequency	Type of Parallelism	Number of processors						
			1	2	4	8	16	30	60
			Time in seconds (speed-up)						
air1	1750	High	358.27			54.92 (7.19)		21.74 (22.30)	18.85 (27.29)
		Low							171.04 (2.13)
	1500	High	771.82			112.30 (7.20)		39.40 (22.89)	33.99 (27.30)
	1250	High	1503.49			209.08 (7.37)		67.54 (24.31)	56.56 (29.58)
	1000	High	3909.95			490.38 (8.06)		155.33 (26.13)	158.14 (25.65)

4.3 Discussion of Results.

We chose the frequency thresholds to be in the boundary region where, for a given number of processors, parallel overhead would be a significant issue, as we wanted to avoid the "easy" cases where the granularity of parallelism would make parallelization clearly efficient. We were not able to measure any cases where the granularity of parallel operations were small enough that the dispatching overhead was significant. For the DTP graph, e.g., [1] shows times for runs down to frequencies of 20 and 10, with run-times nearly 10X those of the smallest frequency shown here (50). As the frequency threshold drops and run-time grows, we expected the use of more processors to be clearly advantageous. In practice, the experimental results show this to be true of high-level parallelism, to a point, and not true of low-level parallelism.

The high-level parallel construct clearly shows considerable benefits, scaling even super-linearly in some cases with more work. However, even given the super-linear speed-ups to 16P, we did not see any benefit at or above 30P. We were not able to isolate the source of this lack of further speed-up. Several causes are possible. We did not use any dplace or cpuset commands, so placement of data within the physical memory of the system could have been poor. Data communication within the shared memory could have been intensive enough that it precludes further speedup; this seems unlikely as the large shared arrays are read-only, so would be cached in each processor. There may be scaling limitations in the underlying run-time system, though it would seem they would have to be `taskq/task` specific, as other OpenMP programs exhibit scaling above 30P on the same system.

The low-level parallel construct, by contrast, in only two cases provides any benefit outside the measurement error of the timings, and at such prohibitive cost (2X speed-up when running on 30 processors) as to eliminate its practical use. Indeed it proves a detriment to performance, compared to the serial time, in most of the cases. We investigated this in some detail. The barrier in the middle of the low-level parallel section might appear to be an obvious source of likely inefficiency, but the output created by the `-openmp_profile` option showed minimal time waiting for barriers. We eliminated parallel loop overhead as the inhibitor by chunking the parallel loop to provide 100X more work per iteration, with an inner serial loop doing each map expansion. This change made no difference. Examination of the parallel and master reduction portions of the code showed that on 1P, the parallel work was taking 430ms and the merge was taking 1ms. On 3P, the parallel work took 450ms (wall-clock) and the merge consumed 150ms. Thus it was obvious why no speed-up was being seen. Each individual call to `register_extension` (which is called from `expand_map` and registers a candidate extension as being possible) was taking longer in the 3P case than the 1P case. Further, it was the parallel regions for subgraphs with large vertex maps that were the source of the poor scaling, due to highly variable (10-100X) times for registering extensions. The routines called by `register_extension` that were showing the high variability had little in-line code and the lengthy delays were due to lengthy delays in calls to `malloc`. These delays were not predictable; *i.e.*, the set of slow iterations was different from run to run. We used `mallopt` to cause `malloc` to allocate memory initially and retain it in user space, to avoid costly kernel calls, but this did not effect the timing. This will need to be understood more fully to resolve the lack of low-level scaling.

5 Conclusions

This work illustrates that, for a graph algorithm with unbalanced parallelism that cannot be predicted in advance, OpenMP and especially the new `taskq/task` constructs provide an effective way of implementing parallelism, both from the points of view of the development work necessary and the resulting scalability. The ability to parallelize the code with minimal changes (~20 directives in a ~15,000 line program) make this a practical approach for researchers wanting one (but perhaps not two) order of magnitude performance improvement with a modest investment of development time. While the code was structured to exploit both high- and low-level parallelism, the low-level parallelism did not exhibit any benefit for the graphs used, for reasons that are poorly understood.

Acknowledgments. This work would have been impossible without the prior work by Michihiro Kuramochi and consultations with him. Even though the UPC port was ultimately not successful, extended discussions with Parry Husbands and Bill Carlson allowed that work to proceed as far as it did. The OpenMP work was greatly accelerated by discussions with John Baron of SGI and Jay Hoeflinger and Grant Haab of Intel, especially concerning the use of the `taskq/task` constructs. The comments of the reviewers improved the paper considerably.

References

1. M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In SIAM International Conference on Data Mining (SDM-04), 2004. http://citeseer.ist.psu.edu/article/kuramochi04finding.html
2. MPI Forum, MPI2: A message-passing interface standard. International Journal of Supercomputer Applications and High Performance Computing 1998; 12:1--299.

3. Devine, K. D., Hendrickson, B. A., Boman, E., St. John, M., and Vaughan, C.: Zoltan: A Dynamic Load Balancing Library for Parallel Applications; User's Guide. Sandia National Laboratories, Albuquerque, NM, (1999). Tech. Report SAND99-1377. Open-source software distributed at http://www.cs.sandia.gov/Zoltan.
4. W.W. Carlson, J.M. Draper, D.E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification, March 1999. Available at http://www.gwu.edu/~upc/pubs.html.
5. Intel C++ Compiler User's Guide (2005), www.intel.com/cd/software/products/asmo-na/eng/compilers/219285.htm
6. Pixar, B. Perens. "Electric fence, malloc debugger." Free software foundation, 1995. http://perens.com/FreeSoftware/ElectricFence/.
7. OpenMP Application Program Interface, version 2.5 (May 2005). http://www.openmp.org/drupal/mp-documents/spec25.pdf.