University of Texas at Austin Institute for Computational Engineering and Sciences (ICES)

GMetis - Xeon Phi

David Pereira
pg22821@alunos.uminho.pt

Rui Brito
pg22781@alunos.uminho.pt

Austin, August 2013

Abstract

In this report, we address GMetis, a graph partitioning application that uses the Metis algorithm. This application uses the Galois framework and has as its goal reducing the total edgecut of a graph partition while keeping the number of vertices balanced between partitions. We show scalability measurements of the application when running it in the brand new Xeon Phi, the first co-processor from the MIC micro-architecture. We also show some modifications done in order to reduce the application's runtime. Last, but not least, we compare GMetis with metis and mt-metis, a parallel version of metis which uses OpenMP, both in terms of runtime and edgecut.

1 Introduction

Metis is actually the most famous graph partitioning algorithm. It was developed by George Karypis[1] at the University of Minnesota, and its goal is to group weighted graph nodes into partitions so that all partitions have similar weight (balanced partitions) and to reduce total edgecut, which is the total weight of all edges between partitions. This algorithm has three major phases: Coarsening, Partitioning and Refinement. All phases of this algorithm are explained in section 2.

The application addressed in this report is *GMetis*, an application that uses the metis algorithm together with the *Galois* framework to produce an efficient parallel version of *Metis*. Our goal was to port GMetis for the Xeon Phi architecture, optimizing its performance, identifying its limitations and bottlenecks, and compare its runtime and edgecut with Metis and its parallel version, mt-metis.

The rest of this report is structured as follows:
The Xeon Phi architecture and features are explained in section 3. while section 5.1 details results and some enhancements made to improve performance on the Xeon Phi. Section 6 presents our final conclusions.

2 The Metis Algorithm

Formally, the metis algorithm consists of three phases. They are as follows:

- Given a graph $G_0 = (V_0, E_0)$:
  - Coarsening:
    - G_0 is transformed into a sequence of smaller graphs $G_1, G_2, \cdots, G_m$ such that $|V_0| > |V_1| > |V_2| > \cdots > |V_m|$
  - Partitioning:
    - A 2-way partition P_m of the graph $G_m = (V_m, E_m)$ is computed that partitions V_m into two parts, each containing half the vertices of G_0
  - Refinement:
    - The partition P_m of G_m is projected back to G_0 by going through intermediate partitions $P_{m-1}, P_{m-2}, \cdots, P_1, P_0$

Visually, this translates into the following scenarios:

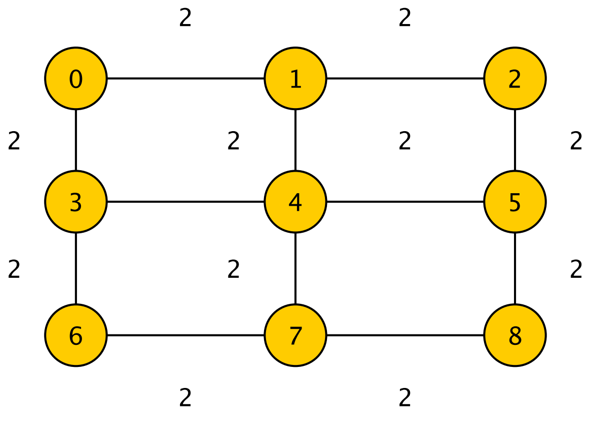

Figure 1: Initial graph

Figures 1 and 2 illustrate the coarsening phase. During this phase, a sequence of coarser graphs is constructed.[2] A coarser graph is constructed by matching neighbour vertices and then contracting the edges. Thus, the edge between two vertices is collapsed and a multinode consisting of those two vertices is created. Also, the edge-cut of a partition in a coarser graph should be equal to the edge-cut of the same partition in the finer graph.[3] This process is achieved in one of two approaches. The first approach consists on finding a random matching and created a multinode with the process described above, while the second approach consists of matching groups of vertices that are highly connected.[3]

[1] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview

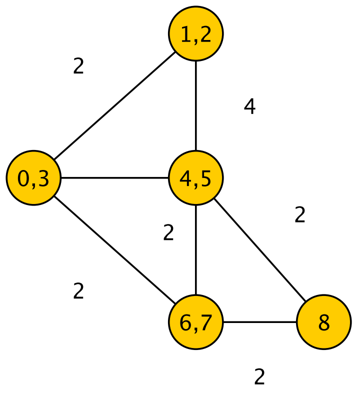

Figure 2: Coarsened graph

Figure 3 displays the partitioned graph, this is the next step in the algorithm. To do this, a Greedy Graph Growing (GGGP) algorithm is used.[3] The goal of this phase, is to compute a high quality bisection (e.g., small edge-cut) of the coarsened graph such that each part contains roughly half of the vertices and edges of the original graph.

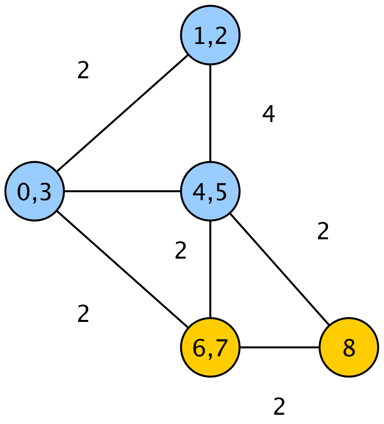

Figure 3: Partitioned graph

Figure 4 shows the results of the refinement phase. During this stage, the partition of the coarser graph is projected back to the original graph by going through the graphs.[3] Once again, the goal here, is to minimize the edge-cut, however, a good balance in the number of vertices assigned to each partition is also very important. Hence, in this final phase, some algorithms use special heuristics to further improve on the balancing achieved.

3 System characteristics

For the purpose of comparing the different applications, measurements were performed in Stampede's co-processors. The co-processors are the new Intel Xeon Phi with 61 cores. Its characteristics are presented in the following table:

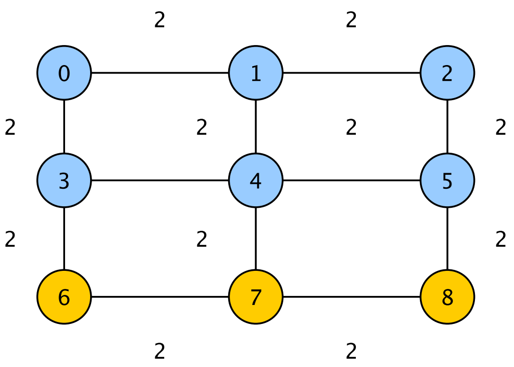

Figure 4: Refined graph

Manufacturer	Intel
Model	Xeon Phi SE10P
μArch	Many Integrated Cores - MIC
Clock freq	1.1 GHz
#CPUs (sockets)	1
#Cores/CPU	61
#Thread/Core	4
L1 cache size/core	32KB
L2 cache size/core	512 KB
Main Memory/CPU	8 GB
Vector width	512 bits

Table 1: Intel Xeon Phi

Each core has support for 4 hyperthreads that shares 512 KB of the second cache level. Thus, the total amount of second level cache memory is more than 30 MB for the entire coprocessor. The more powerful feature on the MIC is a vector unit of 512 bits for each core allowing the execution of 8 double precision or 16 single precision operations in parallel. Globally, the floating peak performance of the Xeon Phi is given by:

- 16 (SP SIMD) x 2 (FMA) x 1.1 (GHZ) x 60 (# cores) = 2112 GFLOPS for single precision arithmetic
- 8 (DP SIMD) x 2 (FMA) x 1.1 (GHZ) x 60 (# cores) = 1056 GFLOPS for double precision arithmetic

Each unit has support for fused multiply add operations (FMA), and together with the vector units, the Xeon Phi achieves theoretical peak performance of more than 1 teraflop.

As can be seen, the Intel Xeon Phi only has 8 GB of memory, thereby limiting large input graphs to run with GMetis. Therefore, all measurements were done using the Western USA road-map (USA-W) as input, which is the largest graph that still fits in memory. This graph contains 6262104 nodes and 15248146 edges.

Apart from the characteristics showed in table 1, there are others that should be mentioned. Figure 5 shows a Xeon Phi core architecture were we can clearly see its 4 hardware threads. Each core contains an in-order dual pipeline which can issue two instructions from the same hardware thread per clock cycle. However, the front-end of the pipeline does not issue instructions from the same hardware thread in consecutive cycles.[1]

This means that if there is only one thread running on a core, for instance T0, this thread may have two instruction issued on the first clock cycle, but none on the second cycle. Thus, the maximum issue rate is only attainable with at least 2 threads per core, so that in each cycle, two instructions may be issued.

Generally, the other two threads have the purpose of hiding pipeline stalls due to memory latency so that the maximum issue rate may be attainable.

Also, it should be noted that the fact that the pipeline issue instructions in-order increases memory related problems. For instance, when there is a stall on a hardware thread, this thread needs to wait for data to be fetched from memory, which means more than 100 cycles. This situation happens every time, which means that only four threads per core might not be sufficient to hide such latency.

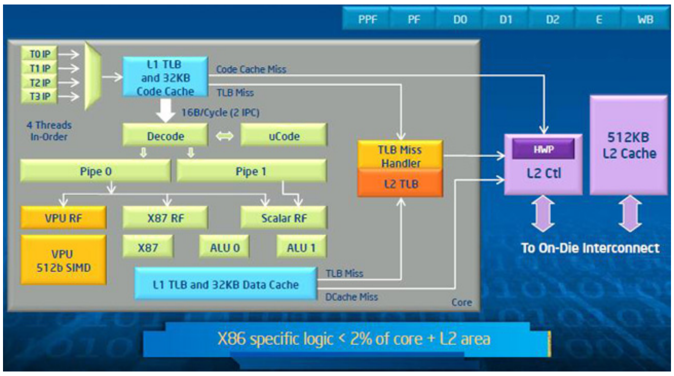

Figure 5: Xeon Phi Co-Processor Core

[2]

4 Metis and Mt-metis

To compare both applications, their runtime and edgecut, we did some scalability measurements. We measured metis runtime on one core of Xeon Phi, and mt-metis for all possible number of threads.

The next figure shows the scalability of Mt-metis on Xeon Phi[3]. Despite the large number of measurements taken, results show a highly irregular plot, with spikes when running mt-metis with a certain number of partitions. The figure also shows speedups relative to the runtime of one thread of mt-metis. We can see that mt-metis scales up to 28 threads, declining rapidly for more than 60 threads. The runtime from 29 up to 59 varies radically. For more than 60 threads, the increasing runtime is mostly due to the clustering and refinement stages. A similar situation happens with gmetis, but as we will see in 5, the costliest phase with a high number of threads is different. We do not fully understand the reason why this happens. With one or few number of threads, the phase that takes more time is Coarsening, followed by Refinement and then Clustering/Partition which is more or less what is expected to happen. But with a high number of threads, this situation changes with the phase that was taking more time before now being the one that takes less time. As we focused only on GMetis, we were not able to tell why this happens. This example is for 128 partitions, but we did measurements for different numbers of partitions, such as 16 and 1024, and all of them shows a similar behavior, reason why only the figure with 128 partition was presented.

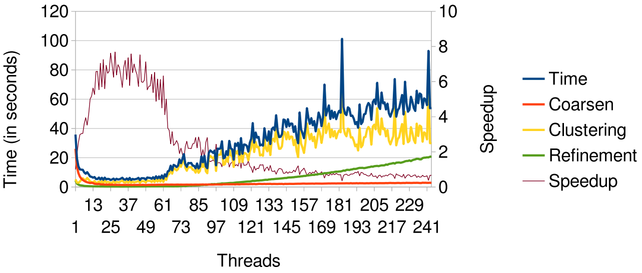

Figure 6: Mt-metis - 128 partitions

5 GMetis

The measurement for GMetis was done in the same way as mt-metis, that is, for all possible number of threads. Figure 7 shows the scalability of gmetis on Xeon Phi for 128 partitions. We can see a similar behavior with mt-metis, with the differences that gmetis scales for a higher number of threads, it does not have a irregular behavior and the performance declines is not so abrupt.

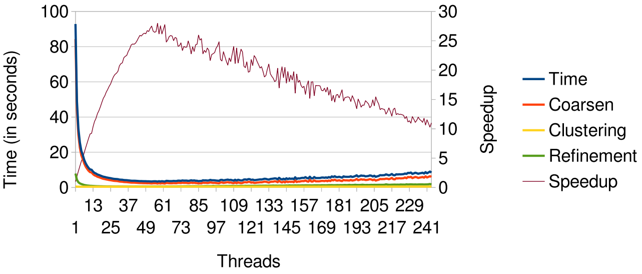

Figure 7: GMetis - 128 partitions

As with mt-metis, the speedup presented in the figure is relative to the runtime of one thread of gmetis when ran on one core of Xeon Phi. Gmetis scales to 70 threads were it attains its maximum peak performance, and decreases linearly with more threads. With more than 70 threads, the global time is more affected by the Coarsening phase. This behavior is different from mt-metis where the Clustering phase was the heaviest. Although gmetis scales better than mt-metis, the latter proves to be really fast for one or few threads as opposed to the former. For instance, if Xeon Phi only provided 28 threads, mt-metis would surely run faster than gmetis.

5.1 Enhancements

Throughout this month, we did some modifications to improve *GMetis*' performance. We started by experimenting how the package mapping is done internally by the *Galois* framework.

Galois has support for *NUMA* systems, thus, this means that it takes advantage of different sockets (packets in Galois terminology) simultaneously.

Before an application starts running, *Galois* parses the cpuinfo file located in "/proc/cpuinfo" to create its mapping, which means that each socket corresponds to a package, or rather, all the cores in a socket belong to the same package.

[3] All measurements were taken with version 5.1.0 and 0.1 of Metis and Mt-metis respectively

This means that *Galois* was assigning a package to the entire MIC co-processor, since it has 60 usable cores in the same "package".

This is especially bad, since the *Galois* framework uses a sort of work stealing mechanism inside each package, to provide good load balance. It works by assigning a master thread in each package, then each thread inside a package can steal work from another inside the same package. In case of aborts, the work is pushed to a stack in the master thread, if it aborts when that master thread pops from the stack, then the work is pushed in another packet's stack. In the MIC, since there is only one packet, this means that all 244 threads were stealing work from each other. Hence, worst case scenario, the code is run serially.

Also, it should be noted that, since *Galois* was not prepared to deal with a processor that supports more than two-way hyperthreading, when using different thread values, some processor cores could have four threads running, while others only one (default mapping).

We changed that, by assigning each hyperthread to its respective core id (which would make a packet). Another reason for this change is so that the mapping could be the most balanced possible (load balanced mapping)[4], for instance, when running the application with 121 threads, with the default mapping, the first 20 cores will run with four threads while the others will only run with one thread. With the load balance mapping, only the last core will run one thread, while the others will run 2 threads. However, this did not achieve any considerable improvements. We also tried assigning each thread to a packet in a round-robin fashion (dense package mapping), but that proved to be ineffective as well (as expected).

To improve gmetis performance, we profiled the application. Unfortunately, Intel VTune only has support for Stampede's hosts, and not for the co-processor. Although, Intel states that profiling and improving an application on the host gives similar improvements on MIC, profiling support for the MIC architecture on *Stampede* would be welcome, as there are key differences in the architectures.

Thereby, we profiled the application with the help of simple timers as well as PAPI, and we found the most time consuming function, which is *findMatching*. This function iterates through the graph's nodes, trying to match each node with one available neighbor.

This function is part of the *Coarsening* phase, together with the *CreateCorseEdge*. The coarsening phase as the objective of coarsening a graph so it can be more easily partitioned. This is done mainly in two phases. The first, findMatching, matches nodes that have not been matched yet, and creates a "supernode" for each pair of matched nodes. The *CreateCorseEdge* creates the edges between "supernodes", merging edges that are shared between the two nodes of the supernode and their common neighbors. The merging operation implies the sum of the weights of these shared edges.

There are different ways to match the graph's nodes. The ones used are "Heavy Weight Match" and "Random Match". The first iterates through each node and matches them with the neighbor whose shared edge has the most weight. The second matches each node with the first neighbor node that has not been matched yet. Nodes that cannot be matched with any other node are not merged. Supernodes are created from these single nodes.

Previously, these two algorithms were used separately, i.e., only one of them was used in the application. Using the two combined proved to be a better solution, as the performance improved, and the edge-cut remained the same. Random Match is used in the first two iterations of the coarsening phase. This improved runtime because, RM computes faster, as it does not need to iterate through all neighbors when there is a node that has not been matched. The algorithm is used only on the first two iterations because the graph is larger on these iterations, and using RM instead of HEM on small graphs did not prove to be any faster and can actually worsen edge-cut.

Has the timers and PAPI were not giving helpful information, we tried to manually fetch some information based on a guess that findmatchting would incur in many misses (mainly because of in-order pipeline). On previous processors that did not contained out-of-order pipelines, software prefetching was almost always useful. We prefetch neighbor nodes, as these were given by an array whose indexes were also given by another array. In this type of situations, prefetch is almost always impossible to happen. However, prefetching did result in improvements. A deeper look into the assembly code generated by the gcc compiler showed that it does not introduce prefetch instructions (even when using __builtin_prefetch). On the other hand, icc does some prefetch. The results, however are quite similar to the gcc ones.

Finally, we also did some tests with different worklist schedulers provided by Galois. AltChunkedLIFO was the fastest and it is actually the most scalable one, which improved a little application's runtime.

The following figure compares the runtime of each application. Metis was executed in one of the Xeon Phi's core, while mt-metis and gmetis were executed for each number of threads, for the three partitions number. For those, the best execution time was chosen. The time of the sequential metis is included to see how much speedup both mt-metis and gmetis obtains for each partition number. We can see that for 16 partitions, mt-metis run faster than gmetis, achieving a total speedup of 24x as opposed to the 17x of gmetis. For the other number of partitions, gmetis is actually faster, achieving a total speedup of 17x and 11x for 128 and 1024 partitions respectively. For the three number of partitions, mt-metis achieves its maximum performance with more or less 30 threads whereas gmetis achieves it with 70 threads.

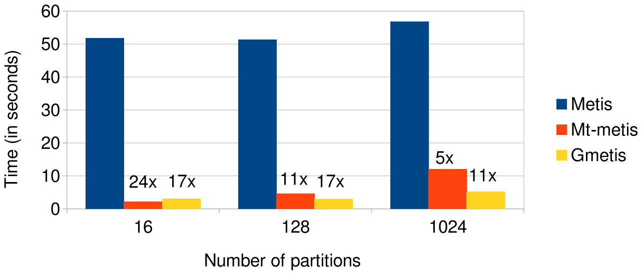

Figure 8: Comparison

We did some extra measurements that allowed us to find that between 45 and 50 partitions, GMetis begins to run faster than mt-metis.

As previously stated, the goal of *Metis* is to partition a graph in a way that the partitions are balanced and the edge-cut between them is the minimized. For this last metric, GMetis performs worse. According to the measurements made and shown on figure 9, *GMetis* edge-cut is always more than two

[4] Although the Xeon Phi contains 61 cores, it should be noted that the operating system also needs to run. Therefore, only 60 cores may be available for computations tasks. With 61, an application can be affected by the operating system noise

times higher than metis and mt-metis. The OpenMP version of metis performs a little bit worse than its original version.

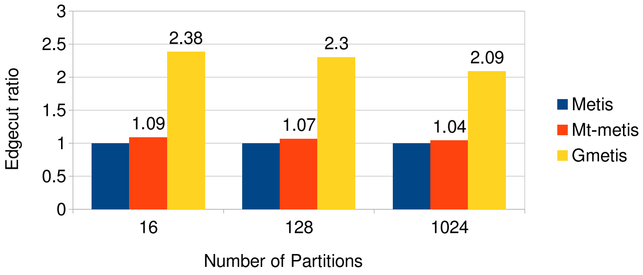

Figure 9: Comparison

6 Conclusion

Results showed that both *Metis* and *Mt-metis* have better edge-cut than *Gmetis* for all number of partitions tested. These two application also run faster than GMetis, for 16 partitions. However, Gmetis's runtime is lower for a high number of partitions, i.e. 128 and 1024 partitions.

Further analysis showed that GMetis start to run faster than mt-metis with more or less 45-50 partitions. However, both metis and mt-metis provides better edgecut. Although the difference of the edgecut between the applications may seem high, it may be not relevant for an higher number of partitions if GMetis computes the partitions much faster than mt-metis. For real applications that uses the metis algorithm, the time consumed calculating the partitions may be more important than a difference of two time for the edgecut.

References

[1] Shannon Cepeda. Optimization and performance tuning for intel® xeon phi^{TM} coprocessors, part 2: Understanding and using hardware events. `http://software.intel.com/en-us/articles/optimization-and-performance-tuning-for-intel-xeon-phi-coprocessors-part-2-understanding`, November 2012.

[2] George Karypis and Vipin Kumar. Parallel multilevel graph partitioning. Technical report, University of Minnesota, 1995.

[3] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, December 1998.