

# GMetis - Xeon Phi

David Pereira

pg22821@alunos.uminho.pt

Rui Brito

pg22781@alunos.uminho.pt

Austin, August 2013

## Abstract

### 1 Introduction

GMetis is a graph/mesh partitioning tool developed using the Galois framework. The Metis algorithm consists of three major phases: Coarsening, Partitioning and Refinement. Our goal was to port GMetis for the Xeon Phi architecture, optimizing its performance, also identifying its limitations and bottlenecks. This report is structured as follows:

Section 2 details the Metis algorithm. Section 3 presents the characteristics of the different system used for testing GMetis. Sections 4 5 present and explain some of the results we got running mt-metis and metis (for comparative purposes), while section 6 details results and some the enhancements made to improve performance on the Xeon Phi. Section 7 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  $G_1, G_2, \dots, G_m$  such that  $|V_0| > |V_1| > |V_2| > \dots > |V_m|$
  - 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  $P_{m-1}, P_{m-2}, \dots, P_1, P_0$

Visually, this translates into the following scenarios:

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

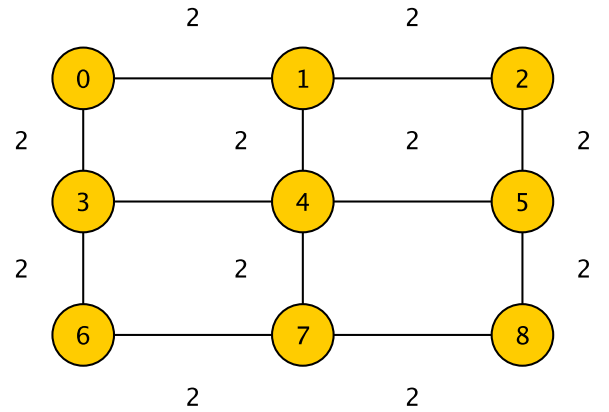


Figure 1: Initial graph

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]

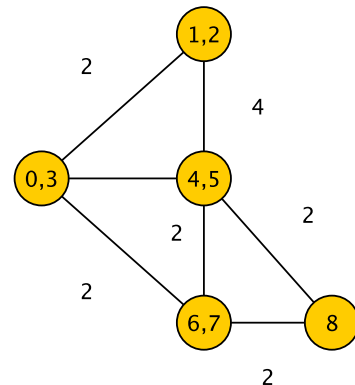


Figure 2: Coarsened graph

Figure 3 displays the partitioned graph, this is the next step in the algorithm. To do this, a Greedy Graph Grow-

ing (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.

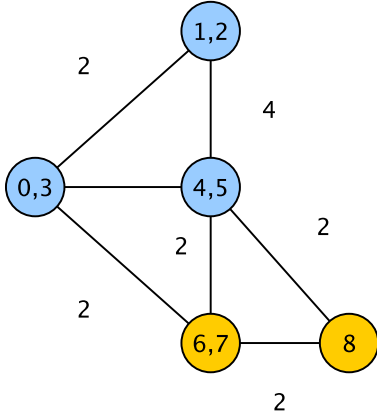


Figure 3: Partitioned graph

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.

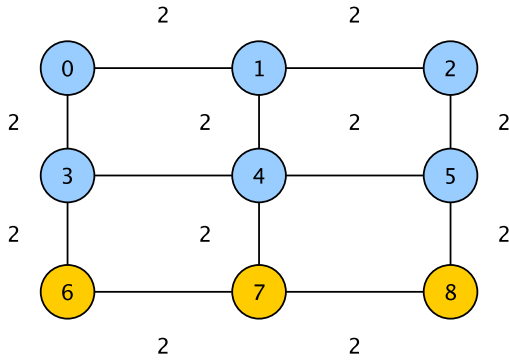


Figure 4: Refined graph

### 3 System characteristics

The measurements were performed in both Stampede's hosts and co-processors. The hosts are comprised of dual Intel Xeon E5-2680, while the co-processors are the new Intel Xeon Phi with 61 cores. Their characteristics are presented in the following tables.

| Manufacturer             | Intel          |
|--------------------------|----------------|
| Model                    | Xeon E5-2680   |
| $\mu$ Arch               | Sandy Bridge   |
| Clock freq               | 2.70 GHz       |
| #CPUs (sockets)          | 2              |
| #Cores/CPU               | 8              |
| #Thread/Core             | 1              |
| L1 cache size/core       | 32 KB          |
| L2 cache size/core       | 256 KB         |
| L3 shared cache size/CPU | 20 MB          |
| Main Memory/CPU          | 16 GB          |
| Vector width             | 256 bits (AVX) |

Table 1: Intel Xeon E5-2680

| Manufacturer       | Intel                       |
|--------------------|-----------------------------|
| Model              | Xeon Phi SE10P              |
| $\mu$ Arch         | 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 2: Intel Xeon Phi

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 with the USA-W roadmap, 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 ??, there are others that should be mentioned. Each core contains a in-order dual-issue 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 the maximum issue rate is only attainable with at least 2 threads per core while the other threads have the purpose of hiding pipeline stalls due to memory latency.

The fact that the pipeline issues in-order instructions increases memory related problems.

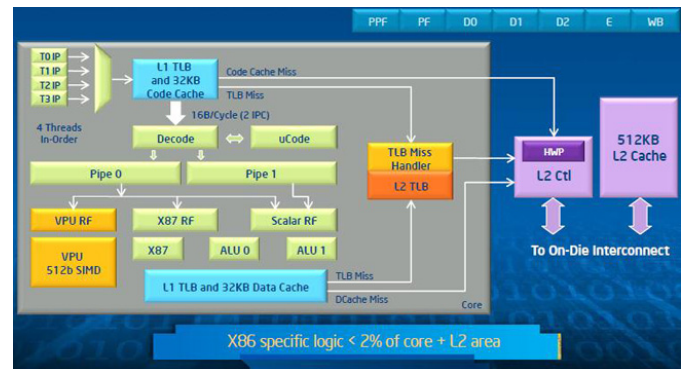


Figure 5: Xeon Phi  $\mu$ Arch

### 4 Metis

The version of Metis we used to perform measurement was 5.1.0

## 5 Mt-metis

The next plot shows the scalability of Mt-metis on Xeon Phi<sup>1</sup>. Despite the large number of measurements taken, results shows spikes when running mt-metis with a certain number of partitions. We can see that mt-metis scales up to 28 threads, declining rapidly for more than 60 threads. This is mainly due to the clustering and refinement stages, since, as we can see from the figure, the coarsening stage is relatively well behaved. The behaviour of the refinement phase is somewhat to be expected, since, with the higher number of threads, as conflicts arise, it is expected for its running time to go up. The problem here, is the Partitioning (Clustering) phase, since its behaviour in unpredictable as the number of threads go beyond 60. We believe the explanation for this behaviour lies the Xeon Phi's architecture, since, as explained above, the maximum issue rate is 2. Combining this with a higher number of threads, the amount of stalls and ensuing context switches is expected to go up. This can be especially bad if the application has a lot of memory random memory accesses, which is the case.

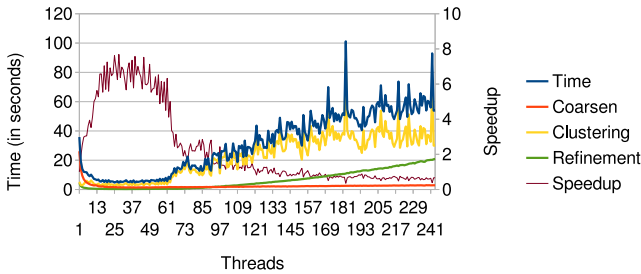


Figure 6: Mt-metis - 128 partitions

## 6 GMetis and the Galois Framework

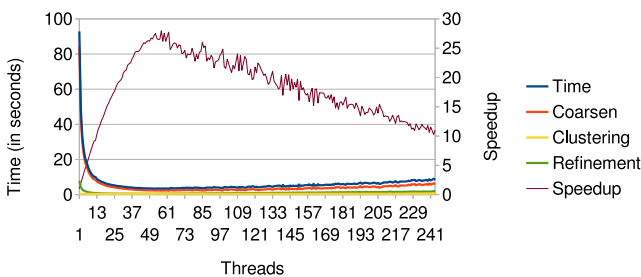


Figure 7: GMetis - 128 partitions

This figure shows the scalability of gmetis on Xeon Phi over the runtime with one thread. This example is for 128 partition, but we did measurements for different number of partitions, such as 16 and 1024. All of them have a similar behaviour.

### 6.1 Enhancements

Throughout this month, we did some modifications to improve *GMetis*' performance. We started by experimenting with

how the package mapping is done internally by the *Galois* framework.

*Galois* has support for *NUMA* systems, thus, this means that it take 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. 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. 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)<sup>2</sup>, 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.

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.

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

<sup>1</sup> All measurements were taken using version 0.1 of mt-metis.

<sup>2</sup> 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.

matches each node with the first neighbor node that has not been matched yet.

Previously, these 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.

A deeper look into the assembly code generated by the two compilers shows that gcc does not introduce prefetch instructions (even when using `__builtin_prefetch`) as opposed to icc that prefetches. The results, however, runtime differences are minimal.

We also did some tests with different worklist schedulers provided by Galois. `AltChunkedLIFO` was the fastest and it is actually the most scalable one.

Some extra measurements allowed us to find that with 45-50 partitions, GMetis start to run faster than mt-metis.

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 partitions, gmetis is actually faster, achieving a total speedup of 17x and 11x for 128 and 1024 partitions respectively.

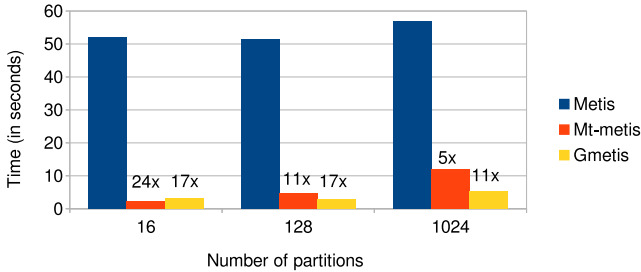


Figure 8: Comparison

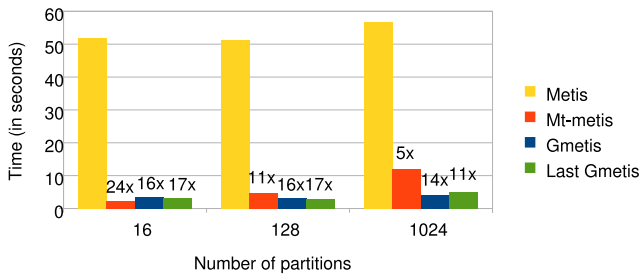


Figure 9: Comparison

As previously stated, the objective of *Metis* is to partition a graph in a way that the partitions are balanced and the edge-cut between them is the smallest possible. For this last metric, GMetis performs worse. According to the measurements made on figure 10, *GMetis* edge-cut is always more than two times higher than metis and mt-metis. The openMP version of metis performs a little bit worse than its original version.

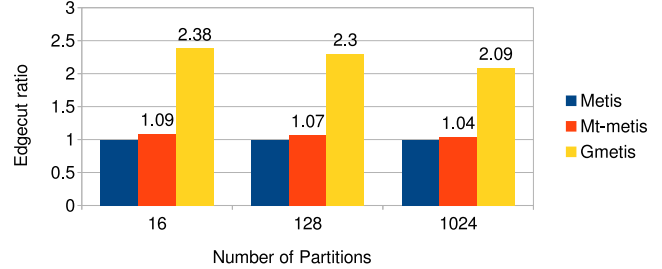


Figure 10: Comparison

## 7 Conclusion

Results showed that both *Metis* and *Mt-metis* have better edge-cut than *Gmetis*. However, Gmetis's runtime is lower for a high number of partitions.

Xeon Phi provides a theoretical performance of 2112 GFlop/sec for double precision arithmetic and 1056 GFlop/s/sec for single precision arithmetic. This values comprises the use of 60 cores since one core is necessary to perform operating system operations.

## References

- [1] Shannon Cepeda. Optimization and performance tuning for intel® xeon phi™ 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.