

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 partitioning application which uses the Galois framework
- Consists of three major phases
 - Coarsening
 - * Find matching nodes
 - * Create Coarse Edges
 - Initial Partitioning (Clustering)
 - Refinement

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 in the finer graph.[3] This process is achieved in one of two approaches. The first approach consists on finding a random

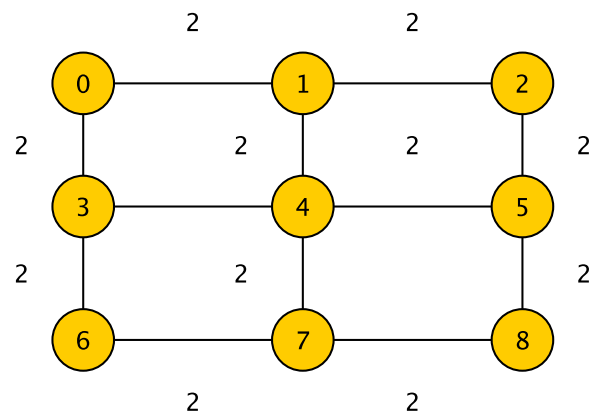


Figure 1: Initial graph

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]

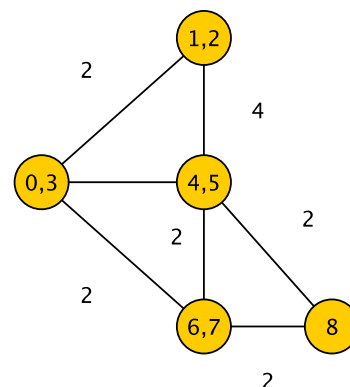


Figure 2: Coarsened graph

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

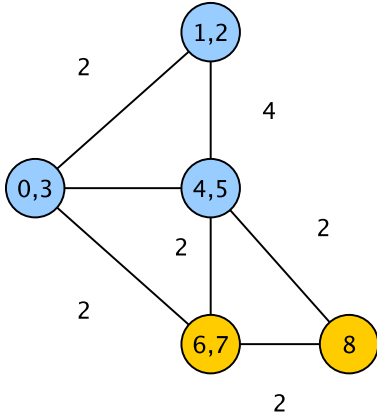


Figure 3: Partitioned graph

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.

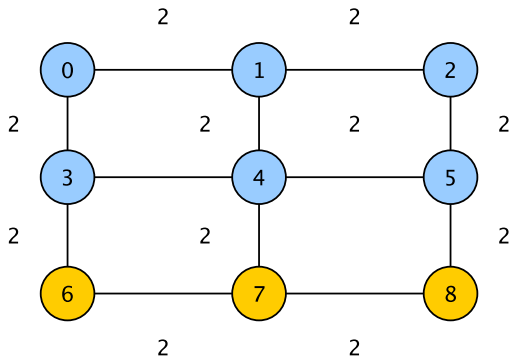


Figure 4: Refined graph

3 System characteristics

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

Manufacturer	Intel
Model	Xeon E5-2680
μ 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
μ 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

Apart from the characteristics showed in table ??, there are others that should be mentioned. Each core contains a 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 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 issue instructions in-order increases memory related problems.

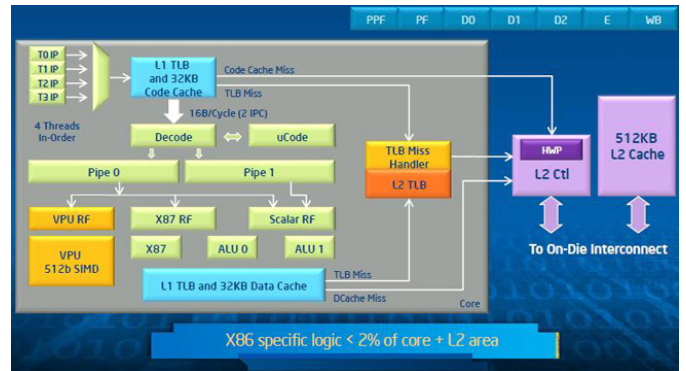


Figure 5: Xeon Phi μ Arch

4 Metis

The version of Metis we used to perform measurement was 5.1.0

5 Mt-metis

We used the 0.1 version of mtmetis.

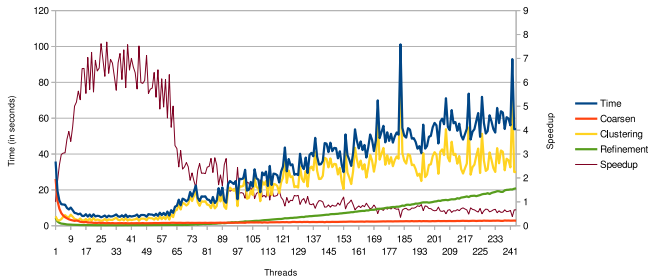


Figure 6: Mt-metis - 128 partitions

6 GMetis and the Galois Framework

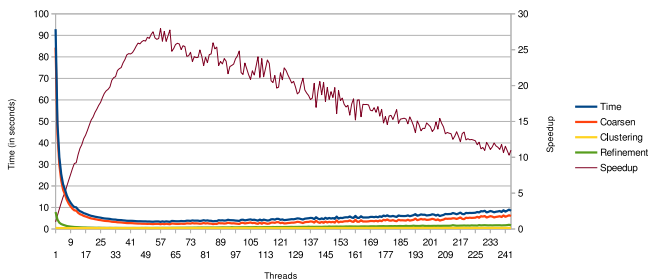


Figure 7: GMetis - 128 partitions

This figure shows the scalability of gmetis on Xeon phi over the runtime with on 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.

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

Galois has support for *NUMA* systems. This means that it can run on different sockets (packets in *Galois* terminology) simultaneously.

Before an application starts running, *Galois* parses the `cpuinfo` file located in `"/proc/cpuinfo"` to create its packages. *Galois* normally assigns a package to each socket. Therefore, *Galois* was assigning a package to the entire MIC coprocessor.

We changed that, but the results did not changed considerably. Another characteristic was that galois was not prepared to deal with processor that support more than hyperthread two thus when using different thread values, some processor cores could have four threads running, while others only one (default mapping). We also changed that so that the mapping could be the most load balanced possible (load balanced mapping). 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.

Although the Xeon Phi contains 61 cores, you should remember that the operating system also needs to run. Therefore, only 60 cores may be available for computations tasks.

Unfortunately, Intel VTune only has support for Stampede's hosts, and not for the coprocessor. Although, Intel states that profiling and improving an application on the host gives similar improvements on MIC, supports for profiling the MIC would

be welcome, as there are important differences in the architectures.

We profiled with the help of simple timers as well as with 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.

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 match 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.

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 edgcut 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 does not proves to be faster and can actually worsen edgcut.

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 oposed to icc that prefetches. The results, however do not differences on runtime.

We also did some tests with different worklist schedulers provided by Galois. `AltChunkedLIFO<8>` was the fastest and it is actually the more scalable one.

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

7 Conclusion

Results showed that both *Metis* and *Mt-metis* have better edgcut 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.