Enabling High-Level Graph Processing via Dynamic Tasking

Maurizio Drocco
maurizio.drocco@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Vito Giovanni Castellana
vitogiovanni.castellana@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Marco Minutoli
marco.minutoli@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

Antonino Tumeo
antonino.tumeo@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

John Feo
john.feo@pnnl.gov
Pacific Northwest National Laboratory
Richland, WA, USA

ABSTRACT

Graph processing is a paradigmatic example of data-intensive computing, in which fine-grained accesses with unpredictable patterns induce irregular and unbalanced workloads. This is exacerbated by large-scale problems that require running on distributed systems, for which no universal solutions are known for solving load unbalance. In this work, we propose dynamic tasking as a way to overcome some typical bottlenecks in task-based runtime systems. As a proof of concept, we implemented the proposed techniques within the tasking system underneath a library of distributed C++ containers. Preliminary experimental evaluation shows consistent scalability, a neat improvement in performance (e.g., 1.5x speedup with respect to the original code over an 8M-nodes graph), and less sensitiveness to parameter tuning.

1 MOTIVATION

Large-scale graph processing naturally induces data-intensive processing, stemming from fine-grained, unpredictable accesses to data. From a computational perspective, this yields irregular and unbalanced workloads, in particular when scaling up to distributed-memory systems such as clusters of multi-core nodes. In this scenario, sustaining high performance requires to understand and control many low-level details of the computation, including memory access patterns and data distribution across the processing nodes. Therefore, designing a programming environment that provides both a high-level Application Program Interface (API) and a high-performance runtime system, that is also able to cope with dynamic workloads such as those induced by graph processing, is a valuable challenge.

To this aim, several task-oriented runtime systems promote massive multi-threading and asynchronous coordination as the primary ways to hide latencies and sustain high performance. However, this poses several challenges in terms of load balancing. In particular, graph applications easily induce burdening workloads, where the distribution of tasks across the processing nodes varies in time, thus producing spikes that result in potential bottlenecks due to load unbalance.

In this work, we advocate *dynamic tasking* as a way to overcome some performance issues that arise in many task-based runtime systems, when dealing with large-scale graph applications. In particular, we propose (1) a novel schema for task allocation, based on dynamic per-thread allocation and an all-to-all recycling network, and (2) a reservation-free remote spawning schema, based on receiver-side buffering and back-pressure feedback/sensing to avoid overflows.

As a proof of concept, we experiment the proposed approach by implementing a variant of the GMT (Global Memory and Threading) runtime system[1] [3] and we exploit it as a back-end of the SHAD (Scalable High-performance Algorithms and Data-structures) library[2] [2]. We measure the performance effects on the well-known graph triangle counting application.

2 METHODOLOGY

In this section, for the sake of simplicity, we first introduce GMT and its current implementation (Sect. 2.1), then we present the proposed proof-of-concept variant, discussing on-node dynamic allocation (Sect. 2.2) and off-node task spawning (Sect. 2.3).

2.1 GMT

In this work, we consider GMT as a paradigmatic task-based runtime system. From the performance standpoint, the key aspect is *fine-grained multi-threading* combined with an *asynchronous execution model.* for latency tolerance. Moreover, *data aggregation* maximizes the exploitation of the available network bandwidth.

On each GMT process, multiple *worker threads* execute user-defined tasks (mapped to lightweight threads), while multiple *helper threads* schedule remote requests (i.e., coming from another process) of task execution. The memory for tasks is allocated concurrently by both worker and helper threads, from a *task allocator* with a custom allocation/recycling logic.

Since the tasking system follows an all-static design, in which all the memory for tasking has fixed size and it is pre-allocated at the beginning of the execution, GMT hardly responds to load unbalance due to variations in time of the task distribution across processes. To avoid performance degradation, it is mandatory to fine-tune the configuration parameters (e.g., size of the tasking memory pool).

SC'18, November 11–16, Dallas, Texas USA
2018.

[1] The current GMT implementation can be cloned at: https://github.com/pnnl/gmt, while the proposed variant can be cloned at: https://github.com/droccom/gmt.
[2] SHAD can be cloned at: https://github.com/pnnl/shad.

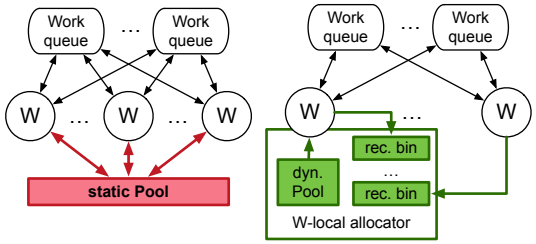

Figure 1: Replacement of static MPMC allocation (left) with dynamic private allocation and SPSC recycling (right).

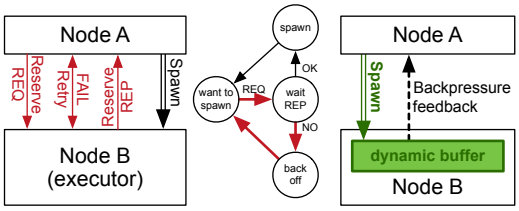

Figure 2: Replacement of distributed remote reservation (left), possibly inducing wait-retry loops (center), with controlled executor-side buffering (right).

2.2 On-Node Dynamic Allocation

In the current allocation schema, worker and helper threads share a single fixed-size Multi-Produced-Multi-Consumer (MPMC) queue (i.e., the task-memory pool), from/to which the memory for tasks is allocated/recycled. In addition to preventing dynamic task allocation, this design may easily lead to performance penalties due to high MPMC contention.

Fig. 1 shows the alternative schema we propose in this work, inspired by the FastFlow allocator [1]. We replaced the single-pool schema with an *all-to-all allocation network*. In our decentralized schema, tasks are allocated by each worker and helper thread from a *dynamic-size private pool*, thus eliminating any source of contention. Tasks are marked with the allocating thread, so that they can be recycled through one-to-one bins, implemented as fast, cache-friendly, Single-Producer-Single-Consumer (SPSC) queues.

2.3 Off-Node Execution

In the current implementation, a *distributed reservation protocol* is used to allocate execution slots remotely each time a running spawns another task that must be executed on a different process. The reservation protocol serves mainly to avoid issuing too many execution requests to processes, but it exposes some relevant drawbacks. As shown in Fig. 2, it may induce high network traffic and get the spawning task trapped into a waiting loop, in case of non-optimal tuning.

In the proposed alternative schema, shown in Fig. 2, tasks are just spawned, in a reservation-free fashion. Instead of making the calling process to wait until some task slots become available at the remote executor side, tasks are possibly buffered at the executor side. To avoid overflows, we employ a lightweight feedback/sensing mechanism for detecting back-pressure.

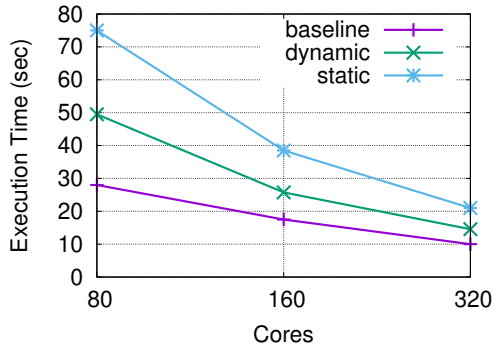

Figure 3: Performance gain by exploiting the proposed GMT variant as SHAD back-end for graph triangle counting.

Cores	2^{23} vertexes	2^{24} vertexes	2^{25} vertexes
80	49.50	108.16	226.91
160	25.75	55.45	122.25
320	14.54	32.16	68.46

Table 1: Strong scaling of exploiting the proposed GMT variant as SHAD back-end for graph triangle counting, for different graph sizes (all the execution times are in seconds).

3 EVALUATION

We performed a preliminary performance evaluation by considering the proposed GMT variant as a back-end of the SHAD library [2] and measuring the improvement in terms of execution times, over graph triangle counting. In our targeted scenario (cf. Sect. 1), SHAD represents a framework for high-level cluster programming, providing a SPMD-free API based on C++ containers, enabling compact and user-friendly representations of complex algorithms.

The experiments have been conducted on a cluster of 16 nodes, each equipped with two Intel Xeon E5-2680 v2 CPUs working at 2.8 GHz (hyper-threading disabled) and 768 GB of memory. As use case, we considered graph triangle counting over different sizes of synthetic Erdös-Renyi graphs.[3]

Fig. 3 shows the comparison among the following variants:

baseline a custom low-level GMT implementation;
static a SHAD implementation with the current GMT back-end;
dynamic the same as static, but with the proposed GMT back-end.

We observe an average performance improvement of about 1.5x speedup with respect to the current GMT implementation. Moreover, although not shown in the figure, we broke the dependency with parameter tuning: while the proposed variant exhibits negligible correlation between configuration parameters (e.g., amount of pre-allocated memory), we observed the performance ranging up to 10x in the original GMT implementations, within the configuration space.

Finally, Tab. 1 shows that strong scaling is sustained by the proposed GMT variant, over synthetic graphs of sizes ranging from 2^{23} to 2^{25} in terms of number of vertexes.

[3]The graphs can be obtained at: https://www.cc.gatech.edu/dimacs10/archive/er.shtml.

REFERENCES

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. 2017. FastFlow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*. Wiley, Chapter 13.

[2] Vito Giovanni Castellana and Marco Minutoli. 2018. SHAD: The Scalable High-Performance Algorithms and Data-Structures Library. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*. 442–451.

[3] Alessandro Morari, Antonino Tumeo, Daniel Chavarría-Miranda, Oreste Villa, and Mateo Valero. 2014. Scaling irregular applications through data aggregation and software multithreading. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 1126–1135.