Parallelizing Irregular Applications through the YAPPA Compilation Framework

Silvia Lovergine*†, Antonino Tumeo*, Oreste Villa‡, Fabrizio Ferrandi†
†Politecnico di Milano, DEIB - Milano, Italy
*Pacific Northwest National Laboratory - Richland, WA, USA
‡NVIDIA - Santa Clara, USA

1. INTRODUCTION

Modern High Performance Computing (HPC) clusters are composed of hundred of nodes integrating multicore processors with advanced cache hierarchies. These systems can reach several petaflops of peak performance, but are optimized for floating point intensive applications, and regular, localizable data structures. The network interconnection of these systems is optimized for bulk, synchronous transfers. On the other hand, many emerging classes of scientific applications (e.g., computer vision, machine learning, data mining) are irregular [1]. They exploit dynamic, linked data structures (e.g., graphs, unbalanced trees, unstructured grids). Such applications are inherently parallel, since the computation needed for each element of the data structures is potentially concurrent. However, such data structures are subject to unpredictable, fine-grained accesses. They have almost no locality, and present high synchronization intensity. Distributed memory systems are naturally programmed with Message Passing Interface (MPI). Moreover, Single Program, Multiple Data (SPMD) control models are usually employed: at the beginning of the application, each node is associated with a process that operates on its own chunk of data. Communication usually happens only in precise application phases. Developing irregular applications with these models on distributed systems poses complex challenges and requires significant programming efforts. Irregular applications employ datasets very difficult to partition in a balanced way, thus shared memory abstractions, like Partitioned Global Address Space (PGAS), are preferred.

In this work we introduce YAPPA (Yet Another Parallel Programming Approach), a compilation framework, based on the LLVM compiler, for the automatic parallelization of irregular applications on modern HPC systems. We briefly introduce an efficient parallel programming approach for these applications on distributed memory systems. We propose a set of compiler transformations for the automatic parallelization, which can reduce development and optimization effort, and a set of transformations for improving the performance of the resulting parallel code, focusing on irregular applications. We implemented these transformation in LLVM and evaluated a first prototype of the framework on a common irregular kernel (graph Breadth First Search).

2. PROPOSED APPROACH

This section briefly introduce GMT (Global Memory and Threading library) [2], the run-time library we use for improving the performance of irregular applications. We then describe the YAPPA compilation framework, mapped on top of GMT, for parallel code generation.

2.1 GMT Library

GMT is a run-time library for irregular applications on distributed memory HPC systems. It provides an API to produce a parallel version of a sequential application. GMT is built around three main concepts: global address space, latency tolerance through fine-grained software multithreading, and aggregation. Fist, GMT implements a PGAS data model, which enables a global address space across the distributed memory of the system, without loosing the concept of data locality. This allows developing the application without partitioning the data set. Parallelism is expressed in form of parallel for constructs. GMT implements a fork/join control model. With respect to SPMD control models, typical of message passing, or PGAS programming models, this model better copes with the large amounts of fine-grained and dynamic parallelism of irregular applications. Second, GMT implements lightweight software multithreading, which allows tolerating latencies for accessing data at remote locations. It provides primitives for atomic operations, which allow to manage synchronization among the nodes. When a core executes a task that issues an operation to a remote memory location, it switches to another task while the memory operation completes, hiding the access latency with other computation. Finally, GMT aggregates the commands directed towards each node to reduce overheads for fine-grained network transactions.

2.2 The YAPPA Parallelizing Compiler

YAPPA extends the LLVM compiler through a set of transformations and optimizations for irregular applications. It targets the GMT run-time, executing on commodity clusters. It takes in input a C/C++ application, manually instrumented by the programmer with synchronization primitives (e.g., atomic addition, compare-and-swap, etc.), and produces a parallel C/C++ version, by instrumenting the LLVM's intermediate representation with GMT primitives. The transformations consists in two steps: data management and loop parallelization. In the first step, YAPPA identifies shared data, and it transform the access to such data in global memory accesses. It also tries to move at the beginning of the loop as many independent memory operations as possible, substituting blocking memory operation primitives with their equivalent non-blocking versions (*unblocking* of memory accesses). Even if GMT has an aggregation mechanism, YAPPA tries to extract as much parallelism as possible (with the unblocking transformation) while at the same time reducing the overhead due to numerous transfers,

by increasing the amount of data per single transfer. In the second step, YAPPA effectively performs the parallelization, by creating the tasks. It extracts parallel loop bodies to generate the task functions. It performs alias analysis to identify incoming arguments, and it allocates proper data structures for them. All the variables, which are defined outside the loop, but used only inside the loop, are *localized* to the corresponding task. Because of the distributed memory architecture, passing parameters to tasks corresponds to memory copies and data transfers, thus localizing variables potentially provides lower communication overheads. Once YAPPA has created the task function, it computes the chunk size (i.e., the number of loop iterations associated to each task). YAPPA also accepts chunk sizes as a command line option. In the first prototype of the compiler, we exploit this option to allow hand tuning of this parameter according to the performance provided by the parallelized program. The application developer can quickly explore several chunk size alternatives in reasonable times through simple compilation scripts. We are currently implementing the support for automatically computing the chunk size according to the irregularity level of each loop. Irregularity analysis will be available in future versions of the YAPPA compiler. The transformation continues with the insertion of the instructions for computing the number of tasks at run-time. The number of tasks spawned by the run-time corresponds to the number of iterations divided by the chunk size, and incremented by one when the number of iterations is not an exact multiple of the chunk size. Finally, YAPPA adds a call to a proper GMT primitive, which manages task functions.

YAPPA parallelizes nested loops by topologically ordering loops according to their nesting level, and by recursively running the parallelization pass on the output of the previous execution. In general, current common parallelizing compilers do not support parallelization of nested loops, because they only look for limited amounts of parallelism, and the target systems do not require large amounts of fine-grained tasks. The user can specify which loops to parallelize by command line option. The approach is equivalent to annotating parallel loops in the program with pragmas, as it is common in solutions such as OpenMP. We also underline that not parallelizing certain nested loops may provide options for further communication optimizations. YAPPA supports also another optimization, dubbed *block-hoisting*. Whenever a loop reads a shared array, YAPPA translates these accesses to get operations. However, if the loop only reads scalar values from the array, sequentially, one iteration after the other, and the loop is not parallelized, this only generates a sequence of very fine grained operations inside the same task. Even if GMT can tolerate data access latencies by switching to other tasks, there still is benefit in aggregating as much communication as possible. In such a case, YAPPA hoists the scalar read operations from the loop and aggregates them in a single get operation, writing to an entire local subarray.

3. EXPERIMENTAL RESULTS

The main benefit provided by YAPPA is a reduction in development time of irregular applications on platforms running GMT. However, in this section we show its effectiveness by applying the transformations on the full queue-based BFS algorithm. We measure the performance of the parallel code produced by YAPPA on a system which emulates a distributed memory design running GMT. The system consists in a quad-processor AMD Opteron 6176SE (codename "Magny Cours") with 256 GB of DDR3 RAM. Each processor hosts 2 dies, and each die features 6 cores and a shared L3 cache of 6 MB, for a total of 48 cores. A core includes a private 512 KB L2 cache and private instruction and data L1 caches of 64 KB each. The processors have a frequency of 2.3 GHz. GMT runs two MPI processes, each one with 15 workers and 15 helper. Each worker hosts up to 1024 fine-grained tasks.

Table 1: BFS performance. (V is the number of edges for the whole graph, E is the average number of edges for each vertex)

#V-#E	Serial	GMT	Sp. GMT	GMT-BH	Sp. GMT-BH
1,000,000-100	1.35	0.52	2.59	0.49	1.06
1,000,000-1,000	9.83	2.53	3.88	2.39	1.06
1,000,000-10,000	95.86	22.72	4.23	21.27	1.07

Table 1 shows the performance (in seconds) of the BFS kernel on the emulation platform, when applying different steps of the YAPPA compilation framework. The first column (*Serial*) shows the serial performance of the code. The second column (*GMT*) shows the performance obtained by only applying loop parallelization for GMT. The third column shows the speedup of the GMT parallel code with respect to its sequential version. The fourth column (*GMT-BH*) shows the performance of the kernel when, beside loop parallelization, YAPPA also applies data management optimizations. The fifth column shows the speedup of the GMT parallel code, when block-hoisting is applied, with respect to the parallel unoptimized code. In particular, for the BFS algorithm, the most important optimization is the block-hoisting of the accesses to the edge list. We parallelized only the outermost loop level of the benchmark. We used graphs of 1 Million vertices where, for each vertex, we changed the number of average edges. The size of the graphs ranges from 0.5 GB to 37 GB. As expected, the speed up increases as the complexity of the graph (number of edges per vertex) increases. The speed up ranges from 2.59 to 4.23. The block-hoisting optimization provides 6 to 7% higher performance. For comparison, the same queue-based implementation, when parallelized through simple OpenMP and run directly on the host platform only provides speed ups in the range of 2.3 times.

4. CONCLUSION

We presented YAPPA, a compilation framework based on LLVM for the automatic parallelization of irregular applications on modern HPC systems. We discussed GMT (a runtime library for the efficient execution of irregular applications on distributed memory architectures) and the YAPPA compilation framework, which builds on top of GMT to produce parallel, optimized irregular applications, starting from a sequential specification. The programmer is only required to manage synchronization by means of atomic operations. The parallel GMT version of the BFS algorithm produced by YAPPA shows an increase in performance.

5. REFERENCES

[1] K. A. Yelick, "Programming Models for Irregular Applications", *in Proc. of Workshop on languages, compilers and run-time environments for distributed memory multiproc.*, Vol. 28, Issue 1, pp. 28–31 1993.

[2] A. Morari and M. Valero, "HPC System Software for Regular and Irregular Parallel Applications", *Proc. of Int. Sym. on Parallel and Distributed Processing Workshops and PhD Forum*, pp. 2242–2245, 2013

[3] E. Gutiérrez, R. Asenjo, O. Plata and E.L.Zapata, "Automatic Parallelization of Irregular Applications", *Parallel Computing*, Vol. 26, Issues 13–14, pp. 1709–1738, 2000.