Productive Data Locality Optimizations in Distributed Memory

Engin Kayraklioglu
The George Washington University
engin@gwu.edu

Tarek El-Ghazawi (Advisor)
The George Washington University
tarek@gwu.edu

ABSTRACT

With deepening memory hierarchies in HPC systems, the challenge of managing data locality gains more importance. Coincidentally, increasing ubiquity of HPC systems and wider range of disciplines utilizing HPC introduce more programmers to the HPC community. Given these two trends, it is imperative to have scalable and productive ways to manage data locality.

In this research, we address the problem in multiple ways. We propose a novel language feature that programmers can use to transform shared memory applications to distributed memory applications easily. We introduce a high-level profiling tool to help understand how distributed arrays are used in an application. As next steps, we are designing a model to describe the implementation of data locality optimizations as an engineering process, which can lend itself to combinatorial optimization. We are also implementing a profile-based automatic optimization framework that utilizes AI to replace the programmer completely in implementing optimizations for distributed memory.

1 BACKGROUND AND MOTIVATION

Programming supercomputers and clusters are significantly difficult compared to personal computers and workstations. Among other issues, this is mainly due to wide range of data access latencies. To utilize the system efficiently, programmers have to choreograph data movements along with computation. Moreover, thanks to ubiquity of HPC systems and increase in problem and data sizes in many fields of engineering, science and data analytics, we are in an era where HPC finds new use-cases. With this trend, more domain scientists and engineers find the need to use HPC services. However, inherent difficulties of programming for such systems pose a significant challenge for such users.

These inherent challenges in programming HPC systems are not new. Programmer productivity has been the major concern in DARPA's HPCS program, under which many programming paradigms such as Chapel, UPC and X10 have emerged. The Partitioned Global Address Space (PGAS) model supported by these languages allow programmers to use *global memory view*, where they can access remote memory directly as if they are accessing local memory. Furthermore, *one-sided communication* reduces the programmer effort by enabling them to use simpler GET/PUT calls instead of SEND/RECEIVE calls widely used in message passing.

On the other hand, such languages fell short of bringing a paradigm shift with regards to how programmers implement their applications. Even with such languages, programmers almost always write code very similar to message passing applications due to high latency [2], where data is aggregated explicitly and moved in as few messages as possible [5]. In this dissertation, we identify the capabilities of different layers of the programming language software stack, and create a novel set of language features and tools to redistribute the labor between the programmer and the programming paradigm.

We first present a novel language feature that significantly reduces programmer effort in transforming shared memory applications into distributed memory applications. Second, we present a high-level data-centric profiler to analyze access patterns to distributed arrays. Third, we summarize our ongoing work that aims to completely relieve the programmer of the data locality optimizations by augmenting our previous work with AI models.

2 LOCALITY-AWARE PRODUCTIVE PREFETCHING SUPPORT FOR PGAS (LAPPS)

We have designed LAPPS [4] as a language feature that the programmer can leverage to implement aforementioned burdensome data locality optimizations. LAPPS, based on observations about pieces of knowledge possessed by different layers of the language stack, augments them with additional capabilities. With LAPPS, the programmer is still in charge of decision making as to what parts of data need to be exchanged across compute nodes. In turn, the language stack handles the arduous tasks of data movement, storage and consistency. This functionality is exposed to the user via *prefetch patterns*.

```
1 A.transposePrefetch();
2 for i in 1..numIter              {
3   forall (i, j) in B.domain do
4     B[i,j] = A[j,i];
5   forall a in A do a += 1.0;     }
```

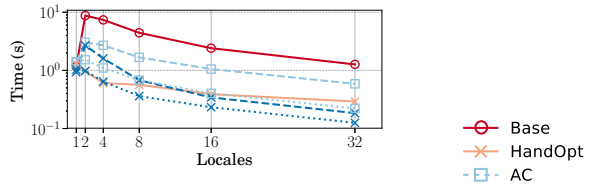
Listing 1: Matrix Transpose Example with LAPPS

Listing 1 shows the transpose prefetch pattern in use with a simple matrix transposition kernel inspired by the Parallel Research Kernels (PRK) [1]. Note that, application logic (lines 2-5) is identical to a shared memory application. The data that is stored in global array `A` is moved according to the transpose prefetch pattern, and the data is stored in internal buffers. Moreover, updates to the input array (line 5) is propagated automatically according to the memory consistency rules of the Chapel language.

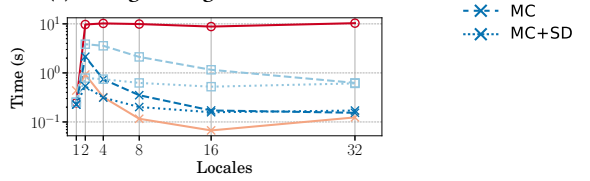
Figure 1 shows the strong scaling performance of LAPPS compared to manual optimization and no optimization on two different interconnects. LAPPS, with some additional common case optimizations enabled, can achieve comparable or better than hand optimized code. Our relevant manuscript contains several synthetic benchmarks, weak scaling study and memory footprint analysis, as well.

3 ACCESS PATTERN ANALYSIS TOOL FOR DISTRIBUTED ARRAYS (APAT)

Optimizations of concern require the programmer to have an understanding of the distribution of the arrays and how they are



(a) Strong Scaling on Infiniband



(b) Strong Scaling on Gemini

Figure 1: PRK-Transpose Performance Results (AC: Automatic Consistency, MC: Manual Consistency, SD: Static Domain Optimization)

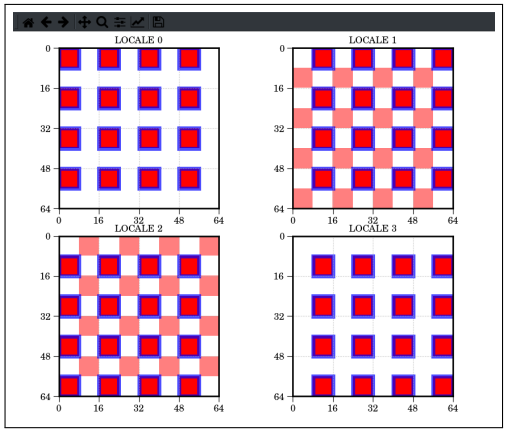


Figure 2: Full Screenshot of HPCC-PTRANS with APAT Visualizer

accessed, whether they use LAPPS or opt for manual optimizations. Therefore, we designed APAT [3], a high-level profiling tool for distributed arrays. APAT logs accesses to distributed arrays in an efficient manner and reports the results in different ways. Implemented in Python, the visualizer uses the matplotlib library to visualize access patterns, as well as the distribution of the array across locales (compute nodes, in this context). Figure 2 shows how HPCC-PTRANS is visualized in APAT. This interface allows basic interaction via the toolbar on top of the screen.

We used APAT to diagnose a load imbalance problem in LULESH which we easily solved by adding only couple of lines leading to %5 performance improvements. In addition, we used APAT to implement manual prefetching in LULESH, leading to more than %30 performance improvement. Our relevant paper also includes performance and storage overhead analysis of APAT.

4 ONGOING RESEARCH

The conclusion of this research revolves around two ideas. First, we are designing a conceptual model which can be used to describe data locality optimizations as an engineering process. This involves clear descriptions of generic subtasks associated with data locality optimizations and capabilities required to perform them. This, accompanied by identification of capabilities of language layers and the programmer will allow us to represent assigning tasks to programmers and the language as a combinatorial optimization problem. We have currently designed the model and defined the capabilities using the fuzzy set theory.

The second idea is to leverage the orthogonality of application logic and data locality optimizations made possible by LAPPS. We are designing an automatic optimization framework that uses APAT to collect profiling data from distributed arrays using very small problem sizes, feeds these data to an AI model as training data, and finally uses this model to optimize applications with LAPPS. Because LAPPS retains the global memory view, this framework can **(1)** tolerate small inaccuracies in predicting the access patterns, **(2)** inject basic method calls to the application to facilitate data movement without changing the application logic that may require thorough static code analysis.

5 CONCLUSION

Our research in the context of this dissertation aims at significantly reducing the programmer effort in programming for distributed memory architectures while achieving scalability. We have designed, prototyped and published two works: a language feature to enable programmers to move data in a very productive way and a profiler for distributed arrays. These can be used in conjunction with one another or in standalone fashion. During the completion phase, we are designing a framework to bring both together to completely relieve the programmer of the burden of data movement. Also, we are creating a generic model to understand data locality optimizations which can be used in designing programming interfaces.

RELATED PUBLICATIONS

[1] Engin Kayraklioglu, Wo Chang, and Tarek El-Ghazawi. 2017. Comparative Performance and Optimization of Chapel in Modern Manycore Architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1105–1114. https://doi.org/10.1109/IPDPSW.2017.126

[2] E. Kayraklioglu and T. El-Ghazawi. 2015. Assessing Memory Access Performance of Chapel through Synthetic Benchmarks. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 1147–1150. https://doi.org/10.1109/CCGrid.2015.157

[3] Engin Kayraklioglu and Tarek El-Ghazawi. 2018. APAT: An Access Pattern Analysis Tool for Distributed Arrays. In *Proceedings of the 15th ACM International Conference on Computing Frontiers (CF '18)*. ACM, New York, NY, USA, 248–251. https://doi.org/10.1145/3203217.3203266

[4] Engin Kayraklioglu, Michael Ferguson, and Tarek El-Ghazawi. [n. d.]. LAPPS: Locality-Aware Productive Prefetching Support for PGAS. *ACM Transactions on Architecture and Code Optimization (TACO)* ([n. d.]), 25. to appear.

[5] Engin Kayraklioglu, Olivier Serres, Ahmad Anbar, Hashem Elezabi, and Tarek El-Ghazawi. 2016. PGAS Access Overhead Characterization in Chapel. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 1568–1577.