Scalable Parallel Solution Techniques for Data-Intensive Problems in Distributed Memory

Nicholas Edmonds  Andrew Lumsdaine
Indiana University
150 S. Woodlawn Ave
Bloomington, IN 47405, USA
{ngedmond,lums}@cs.indiana.edu

Abstract
Computational science and engineering has traditionally been comprised of linear algebra problems induced by large systems of PDEs. These problems often possess underlying natural locality which can be determined analytically, often via static analysis. An increasingly important class of data-intensive applications is emerging that lacks this natural, or domain-induced, locality. Rather than being analytically deducible, the dependency structure of this class of computations is determined by the input data itself. This data-carried dependency structure is expressed at run time and offers limited opportunities for static analysis. These computations benefit from a variety of fine-grained and dynamic solution techniques well suited to the irregular and non-local nature of these problems.

1. Introduction
Traditional compute-intensive applications, such as those based on discretized systems of PDEs, possess natural locality due to the local nature of the underlying operators. This natural locality allows the dependency structure of the computation to be largely determined before any portion of the computation occurs. Because locality can be determined analytically at compile-time, the granularity of the application can be coarsened through static analysis and the grouping of computations with shared data dependencies.

This class of problems often possesses good separators in the dependency graph which allows for straightforward parallelization using SPMD techniques such as the coarse-grained BSP [4] "compute-communicate" model.

Data-intensive applications, such as those based on graphs differ from compute-intensive applications in a number of important ways. Rather than possessing natural, domain- or operator-induced locality these computations are highly non-local. The dependency structure of the computation is irregular and embedded in the input data itself. This means that rather than having dependency information available at compile time and being able to perform static analysis and optimization on it, the dependency structure of the computation is discovered dynamically at run time. Finally, because dependency information is discovered at run-time, work grouping or coalescing cannot be performed at compile-time and thus the natural granularity of the computation is fine-grained.

2. Expressing Fine-Grained Applications
Coarse-grained BSP solutions often group operations which do not share computational dependencies in order to coarsen the granularity of an application and achieve better performance on real hardware. This approach is reasonable in applications with a dependency structure which can be determined analytically. In data-intensive applications with run-time-discovered dependency structures this approach can artificially extend the critical path of the application however. It is thus desirable to express applications at the finest level of granularity possible. Coarsening may still be necessary at run-time, but by providing the run-time mechanism that performs this coarsening with the full, fine-grained dependency graph of the application we enable it to make the most effective decisions about how to perform this coarsening. In order to expose maximum concurrency, expressing applications as independent, asynchronous collections of (possibly dependent) tasks has proven to be an effective programming style. Maintaining data consistency in the presence of asynchronous, concurrent tasks requires supporting transactional semantics on the underlying application metadata that these tasks manipulate, which is a well studied research area [2, 3]. This requirement for transactional semantics precludes the use of most "one-sided" or passive-target RMA approaches but can be incorporated into Active Message-based solutions [5].

3. Run-Time Support
Executing fine-grained applications at their natural level of granularity on modern HPC hardware has a number of problems. HPC hardware has largely been designed to solve traditional coarse-grained, compute-intensive problems. One of the most important manifestations of these design goals with respect to fine-grained applications is the fact that modern high-performance networks become injection-rate limited before they become bandwidth limited in the presence of large numbers of small messages. Secondly, in data-intensive applications where the dependency graph has no good separators, communication patterns are likely to be dense. In compute-intensive applications the number of peers a single processing element (PE) communicates with is often constant or logarithmic due to the natural locality of the underlying operators. In data-intensive applications the number of peers can easily be linear in the number of PEs. While modern networks support collective all-to-all communication relatively well, irregular exchange of small messages between $O(P^2)$ PEs is less well supported for large $P$ and incurs significant per-connection overhead to manage the $P^2$ connections.

The aforementioned limitations of modern HPC networks imply that in order to efficiently map fine-grained expressions of data-intensive problems to current hardware, a run-time layer which can dynamically transform the application expression to an efficient implementation is required. Performing software routing on a sparser virtual topology allows the number of peers each PE communicates with to be reduced from $O(P)$ to $O(\log P)$ or even $O(1)$. Coalescing small messages into larger messages which...
better exploit network bandwidth increases latency but may yield better performance in bandwidth-bound regions of the application.

4. Applications
We now demonstrate the application of the programming and execution model (briefly) described above, to an application within the data-intensive domain. Breadth-first search (BFS) is a simple graph kernel which produces a breadth-first numbering of the vertices of the graph starting with a source vertex \( s \). We assume a vertex distribution of the adjacency list (row-wise distribution of the adjacency matrix).

Algorithm 1: Coarse-grained BFS

```
Input: Vertex \( s \), \text{neighbors}(v) \) a function returning the neighboring vertices of \( v \), \( Q_0, Q_1 \) distributed queues
Output: \( d[v] \) the bfs distance of \( v \) from \( s \)
1 \( \forall v \in V: d[v] = \infty; \)
2 \( d[s] \leftarrow 0; \)
3 \( Q_0, Q_1 \leftarrow 0; \)
4 \( \text{enqueue}(Q_0, \{s, 0\}); \)
5 while \( Q_0 \neq \emptyset \) do
6     while \( Q_0 \neq \emptyset \) do
7         \( u, \text{dist} \leftarrow \text{dequeue}(Q_0); \)
8     if \( \text{dist} < d[u] \) then
9         \( d[u] = \text{dist}; \)
10     foreach \( v \in \text{neighbors}(u) \) do
11         \( \text{enqueue}(Q_1, \{v, \text{dist} + 1\}); \)
12     \end{algorithm}

In a traditional SPMD or BSP-style implementation such as Algorithm 1: we expand one level of the BFS (line 7), then communicate all the vertices in the next level (line 14), and repeat until the next level is empty (line 6). This approach has the effect of producing large blocks of data to communicate collectively, but fails to capture the fine-grained dependency structure of the application and thus does not express the full concurrency available.

Algorithm 2: \text{discover}(v, d_v) \) – active message handler for vertex discovery

```
Input: \( v \) the vertex to discover, \( d_v \) the tentative distance of \( v \) from \( s \), \text{neighbors}(v) \) a function returning the neighboring vertices of \( v \)
1 \( \text{if } d_v < d[v] \) then
2     \( d[v] \leftarrow d_v; \)
3     foreach \( u \in \text{neighbors}(v) \) do
4     \( \text{discover}(u, d_u + 1); \)
```

In the fine-grained expression of the algorithm the core operation, discovering new vertices (Algorithm 2), is expressed as an independent asynchronous operation which may be invoked locally or remotely, and may have many instances executing concurrently. The main body of the algorithm (Algorithm 3) simply discovers the source vertex (line 3) and waits for the chain of executions of Algorithm 2 this triggers to complete (line 4). There are a number of assumptions implicit in this formulation of BFS. First, because the same method is invoked to discover a vertex regardless of whether the vertex is local or remote with regard to the discovering process, some method of resolving vertices to address spaces must exist. As previously discussed, for performance reasons it is also assumed that an underlying run-time layer coalesces these individual method calls bound for remote processors into larger messages sufficient to overcome the injection-rate limitations of the network, and optionally routes them through a virtual topology to minimize connection overhead.

5. Results
We have employed the techniques described above to implement a generalized active message library (AM++ [6]) capable of performing the runtime transformations described. Comparing the performance of traditional BSP-style formulations of graph algorithms in older versions of the Parallel Boost Graph Library [1] with asynchronous formulations of the same algorithms implemented using AM++ demonstrates markedly improved performance (see Figure 1). Additionally, by expressing algorithms as collections of asynchronous, concurrent tasks we simplify the task of leveraging thread-level parallelism as well as process-level parallelism.

![Figure 1. Strong scaling performance of Parallel BGL and AM++-based implementations of parallel BFS (2^{27} vertices and 2^{29} edges).](http://www.osl.iu.edu/research/pbgl)

References

generalized active message framework. In Parallel Architectures and
Compilation Techniques, September 2010.