Latency Hiding and Performance Tuning with Graph-Based Execution

Pietro Cicotti
San Diego Supercomputer Center
University of California, San Diego
9500 Gilman Dr., San Diego, California
92093-0505
pcicotti@sdsc.edu

Scott B. Baden
Dep. of Computer Science and Engineering
University of California, San Diego
9500 Gilman Dr., San Diego, California
92093-0505
baden@cse.ucsd.edu

ABSTRACT

In the current practice, scientific programmer and HPC users are required to develop code that exposes a high degree of parallelism, exhibits high locality, dynamically adapts to the available resources, and hides communication latency.

Hiding communication latency is crucial to realize the potential of today's distributed memory machines with highly parallel processing modules, and technological trends indicate that communication latencies will continue to be an issue as the performance gap between computation and communication widens. However, under Bulk Synchronous Parallel models, the predominant paradigm in scientific computing, scheduling is embedded into the application code. All the phases of a computation are defined and laid out as a linear sequence of operations limiting overlap and the program's ability to adapt to communication delays.

In this paper we present an alternative model, called Tarragon, to overcome the limitations of Bulk Synchronous Parallelism. Tarragon, which is based on dataflow, targets latency tolerant scientific computations. Tarragon supports a task-dependency graph abstraction in which tasks, the basic unit of computation, are organized as a graph according to their data dependencies, i.e. task precedence. In addition to the task graph, Tarragon supports metadata abstractions, annotations to the task graph, to express locality information and scheduling policies to improve performance.

Tarragon's functionality and underlying programming methodology are demonstrated on three classes of computations used in scientific domains: structured grids, sparse linear algebra, and dynamic programming. In the application studies, Tarragon implementations achieve high performance, in many cases exceeding the performance of equivalent latency-tolerant, hard coded MPI implementations.

1. INTRODUCTION

In supercomputing systems, architectural changes that increase computational power are often reflected in the programming model. As a result, in order to realize and sustain the potential performance of such systems, it is necessary in practice to deal with architectural details and explicitly manage the resources to an increasing extent.

Hiding communication latency is crucial to realize the potential of today's distributed memory machines with highly parallel processing modules. While the number of cores per processor continues to increase, specialized processors (often called accelerators) are becoming a common attribute of high performance computing systems[1]. High core counts and accelerators contribute to an increase in the relative performance gap between computation rate and data transfer rate. In addition, the finer granularity required to leverage the parallelism in hardware makes application performance very sensitive to data transfer latency, a slowly improving hardware characteristic [19]. However, parallel programming models still lack high level abstractions for hiding communication latencies.

In the classical Bulk Synchronous Parallel (BSP) programming model, the dominant model in parallel scientific applications [17], processes execute in parallel operating on a partition of the data and alternating a computation phase with a synchronous communication phase. The underlying parallel computer is a set of processors with local memory connected by a router. BSP has been labeled a *bridging* model to emphasize the intent of defining a unified model driving both software and hardware design. In fact, BSP captures the essence of both distributed memory machines and data-parallel algorithms. Among others, the Message Passing Interface (MPI) and Unified Parallel C (UPC), two well known parallel programming models, are intrinsically BSP models.

While characterized by different design and implementation choices, all BSP models preserve the same computation structure with a communication phase that is part of the control-flow and that lies on the critical path. Despite numerous research efforts [3, 5, 9, 24], in BSP models there is no widely adopted solution to hide latency other than to *manually* develop split-phase communication code. The idea of split-phase communication is simple: initiate the communication as soon as possible (first part of the communication phase) and wait for completion only when required by the implicit data dependencies (second part of the communication phase). Though simple as an idea, split-phase communication requires considerable programming effort because it involves extensive code restructuring. In addition, the restructured code may exhibit a locality

[1]Including the system ranked first, four of the 10 most powerful systems in the world according to the Top500 project use accelerators [27]

oblivious memory access pattern.

Data-driven programming models, like dataflow and actors, show promise as a way to expose a high degree of parallelism and automatically achieve overlap. In fact, the ability to achieve overlap in data-driven algorithmic formulations has been demonstrated in ad-hoc implementations of linear algebra kernels [11]. However, current data-driven programming models fail to expose and express the underlying communication pattern and do not present users with high level abstractions that support latency-hiding algorithms.

Large-grain dataflow languages gained traction recently as multi-core architectures pervaded computer architectures. Cilk, for example, is a multithreaded language for task-parallelism [23]. Internally, Cilk activates tasks by maintaining a dependency graph. However, Cilk is intended for shared-memory architectures and does not support performance optimizations. In particular, Cilk is locality oblivious and does not enable overlap of communication with computation. Similarly, StarSS is a directive-based model for task parallelism on shared memory architecture, but does not address and distributed memory architecture and communication cost optimizations [21].

Charm++ is an object oriented parallel language that implements an actors model [15]. In Charm++ actors are special objects, called *chares*, with *entry* methods supporting *Asynchronous Remote Method Invocation*. Entries are like communication primitives: when an entry is invoked, the underlying run-time system creates a message that is sent to the destination chare. In Charm++ execution is virtualized [16] in the sense that chares execute like virtual processes managed by the underlying run-time system. The run-time system also manages communication and it can overlap communication with computation. However, the dataflow structure is embedded in the control flow and it is unveiled only during program execution.

We present Tarragon, a dataflow programming model for latency-tolerant scientific computations [20]. Tarragon supports a task-dependency graph abstraction in which tasks, the basic unit of computation, are organized as a graph according to their data dependencies. Tasks communicate by transferring data along the edges of the graph and the underlying run-time system manages dataflow execution semantics and data motion. In addition to the task graph, Tarragon uses metadata abstractions to express locality information and scheduling policies.

We applied Tarragon to three classes of computations used in scientific domains: structured grids, sparse linear algebra, and dynamic programming [7, 8]. For each of the above classes, we present the implementation of an application using Tarragon, a performance study for the application, and a comparison to an equivalent hand-coded latency-tolerant MPI implementation.

2. PROGRAMMING MODEL

2.1 Overview

Tarragon is based on a dataflow abstraction. The abstraction hides most of hardware and software low-level details while it requires programmers to decompose the problem into tasks and express dependencies between the tasks. Exposing parallelism is still the programmer's duty, as in most widely adopted programming models, although parallel execution and concurrency are implicit and their control transparent to the programmer.

Tarragon's programming model is a coarse-grain dataflow model. In classical dataflow models a program implicitly defines a graph in which nodes are instructions and edges connect nodes such that the result of an instruction becomes an operand of another instruction [6, 12–14]. The *firing rule* is straightforward: an instruction executes when all its operands are available. Data is transferred along the edges, rather than stored in memory, and there is no program counter as the graph execution manages control flow. In Tarragon the nodes of the graph are coarse-grained objects, called *tasks*. Tasks of arbitrary complexity can be defined and can preserve their state across executions. The *firing rule* is user-defined and the data is stored in memory. The edges carry data in the form of *messages*.

2.2 Execution Model

Tarragon's programming model executes under a three-layer control structure. The three levels in Tarragon are the *task level*, the *graph level*, and the *control level*. The task level represents the instructions within a task. The graph level controls task execution according to dataflow semantics. Finally, the control level controls the graph abstractions and is responsible for creating and executing graphs.

The control level initializes Tarragon's run-time system (RTS), creates a graph, and launches the graph's execution. The RTS supports the graph level of execution. Once the tasks of the graph are defined and connected, execution is transparently carried out by the RTS, which manages task scheduling and data motion, orchestrating the dataflow abstraction. At the graph level, the order in which tasks are executed is constrained by dependencies, but among partially ordered tasks[2], the exact ordering is determined by Tarragon's RTS. As dependencies are satisfied, tasks become ready to execute and eventually they are executed by the RTS. When multiple tasks are ready for execution, they run in parallel as available resources allow. Finally, at the task level, tasks execute as virtual processes, unaware of the underlying levels. A task becomes ready to execute when its firing rule is satisfied. Once a task begins executing, it runs to completion. It cannot be preempted and it cannot wait for communication events. These conditions simplify the task code which can be designed as a self-contained sequential program, free of scheduling and concurrency concerns.

Users define the core of the computation within a task, specifying firing rules and the operations that a tasks carries out. A task is therefore characterized by its virtual methods and that the application programmer extends to define a concrete task subclass.

The state of a task regulates the interaction with the RTS. A task is a state machine whose transitions are triggered by method invocation. The RTS inspects the *state* attribute, encoding the observable state of the task, and invokes initialization and execution accordingly. Figure 1 illustrates the states of task, the transitions between states, and the corresponding methods causing the transitions.

2.3 Communication Model

In Tarragon, communication between tasks is expressed and enabled by connecting tasks with directed edges. During graph execution, tasks can move data along the edges of the graph. The graph is an explicit representation of the communication pattern.

[2]A partial order on a set defines an ordering on the elements although not all the elements of the set are comparable; that is, the order relation is not defined on all the pairs of elements.

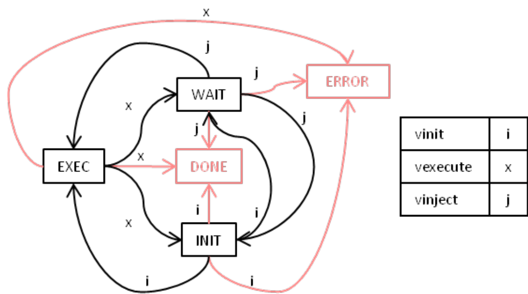


Figure 1: States and transitions. The lines denote state transitions between the possible states of a Task. The labels denote the method causing the transition. Notably, ERROR and DONE are reachable from all the other states and are final states.

Communication is one-sided in the sense that the destination task is not actively involved in the communication. As with Active Messages [25], data arrival triggers a handler function execution. The handler injects data into the task and triggers task readiness according to the firing rule that it encodes. Since sending data is an asynchronous operation and there is no receive operation, a task never blocks on communication and it is therefore guaranteed that, when executing, tasks are actively computing. In this way processor virtualization is implemented without preemption yet it is guaranteed that cores are utilized efficiently.

3. APPLICATIONS

3.1 Jacobi Iterative Solver

The first model computation considered is an iterative solver for the Poisson's equation in a three-dimensional domain with *Dirichlet* boundary conditions. Poisson's equation, as shown in Equation (1), is a PDE that expresses a potential function, here denoted by u, in terms of a known source function, here denoted by v, on an open region $\Omega \in \mathbb{R}^3$. The value of the potential on the boundary is given by a known function f.

$$\begin{cases} \Delta u = v \; in \; \Omega \\ u = f \; on \; \partial\Omega \end{cases} \quad (1)$$

The solver implements Jacobi's method. The space is discretized and represented by a Cartesian grid (uniform spacing with distance h between consecutive points is assumed) and two copies of the grid are stored to support out-of-place updates. Then, a centered 7-point finite-difference stencil is applied to each grid point in order to produce a new approximate solution in each iteration:

In SPMD formulations the grids are partitioned and the partitions mapped each one to a different process. In addition to the boundary values, additional grid points are necessary for the calculations on the internal boundaries. Such additional points, usually referred to as *ghost cells*, are copies of points that belong to neighboring processes. The values of ghost cells is refreshed between iterations.

Four different variants are examined and compared in the remainder of this Section: Synchronous, Asynchronous, BGraph, and Graph.

The Synchronous variant is an MPI encoding. Each MPI process owns a partition of the grid and is responsible for the updates. Ghost cells are exchanged by using non-blocking communication primitives, to improve communication efficiency, but processes synchronize on communication waiting immediately for completion.

The Asynchronous variant is a split-phase reformulation of the Synchronous variant. In addition to using non-blocking primitives, the computation is split in two phases: one to compute points in the inside of the local grid, and one to compute points on the surface, which are the points whose new values depend on the ghost cells. Communication is initiated as before, but there is no immediate wait; rather, the relaxation of the inner points is done first. Then, processes wait for communication to complete. Finally, the points on the surface of the local grid are updated using the ghost cells received. Updating the surface takes 6 additional groups of loops (only two nesting levels this time) to update the points on each face.

The Graph and BGraph variants are implemented with Tarragon. In both variants, tasks are the equivalent of processes in the MPI models with the difference that, in the ideal decomposition, the number of tasks exceeds the number of processor cores (over-decomposition). In addition, tasks do not wait on communication and Tarragon manages execution and communication according to the semantic of the graph. The algorithms are fundamentally asynchronous and differ from each other because BGraph, as well as the MPI variants, implements cache blocking. Consequently, BGraph uses a lower number of tasks and still achieve high locality whereas Graph does not employ explicit cache blocking and simply relies on over-decomposition to achieve the same effect. In addition, since a relatively small number of local tasks means a smaller number of *internal* tasks, BGraph uses a rectilinear blocking schema that allows for blocks with different sizes. With this schema, tasks with off-node edges are associated to smaller blocks to ensure that enough computation is left to be overlapped to communication.

With over-decomposition, mapping can be seen as a coarser level of blocking. From this perspective, a mapping is a partition of the set of tasks where each partition represents a process. To reduce off-node communication and to equally distribute the tasks, BGraph and Graph use a regular three-dimensional blocking schema to map tasks to nodes. The graph is completed by connecting tasks along the Manhattan directions.

3.1.1 Performance Evaluation

Weak scalability reproduces the scenario where users employ a larger set of resources to solve a larger instance of the problem or, as is often the case in scientific computing, solve the same problem at a higher resolution. The first set of experiments conducted on Abe is a weak scaling study to compare the four implementations. The measure of comparison is the achieved GFLOPS rate.

As shown in Table 1, the problem size ranges from 512^3 to 3072^3 running on 8 to 2048 cores, respectively. These configurations maintain approximately 16M points per core and complete 50 iterations in approximately 30 seconds. From the table it can be observed how the Asynchronous variant suffers degraded performance as a result of the poor locality of split-phase coding. It can also be observed that in all the configurations tested, the two Graph variants perform better than the Synchronous version. The speedup enjoyed is the result of the reduced communication cost that lies on the critical path. In fact, the speedup achieved (the seventh column reports the speedup of Graph versus Synch) is proportional to

Table 1: Weak scaling on Abe. The Table shows the performance, measured in GFLOPS, of the four variants: Synchronous (S), Asynchronous (A), BGraph (BG), and Graph (G). The Table also shows the percentage of time spent communicating in Synchronous (Comm), the speedup of Graph over Synchronous (S/G), and the percentage of the communication time that is hidden in Graph (Hidden).

Problem Size	Cores	S	Comm	A	BG	G	S/G	Hidden
512^3	8	1.5	7%	1.4	1.6	1.6	1.06	74%
640^3	16	3.1	4%	2.9	3.2	3.2	1.04	93%
800^3	32	6.0	7%	5.7	6.2	6.3	1.05	74%
1000^3	64	11.6	7%	11.3	12.4	12.5	1.08	98%
1200^3	128	23.6	8%	22.4	24.0	24.9	1.06	64%
1680^3	256	47.3	6%	45.7	49.7	49.8	1.05	83%
2000^3	512	93.4	7%	90.2	99.2	99.3	1.06	87%
2432^3	1024	185.7	9%	175.0	197.8	197.6	1.06	68%
3072^3	2048	372.5	8%	364.7	390.7	394.6	1.06	70%

Table 2: Strong scaling on Abe. Performance, measured in GFLOPS, of the four variants: Synchronous (S), Asynchronous (A), and Graph (G), with the percentage of total running time spent communicating in Synchronous (Comm), the speedup of Graph over Synchronous (S/G), and the percentage of the communication time that is hidden in Graph (Hidden).

Cores	S	Comm	A	G	S/G	Hidden
256	47.3	9%	45.6	49.7	1.05	57%
512	93.8	9%	83.2	99.4	1.06	63%
1024	174.8	15%	127.4	197.2	1.12	78%
2048	312.8	23%	266.8	391.9	1.25	86%

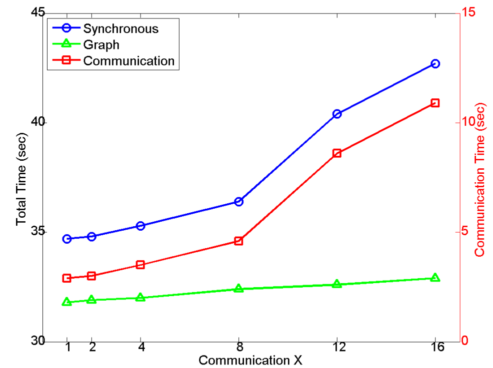


Figure 2: Effect of increased communication cost in Jacobi solver.

the communication cost (the cost of communication is measured on the Synchronous version by shutting off communication). Finally, BGraph and Graph achieve similar performance proving that, in this case, over-decomposition increases locality and obviates the need for cache blocking optimizations.

Strong scaling experiments show that this is the case. Table 2 gives the detailed results, including the performance of BGraph, for a 1600^3 problem on 256, 512, 1024, and 2048 cores. For Synchronous, when the number of cores increases the gap between ideal and measured performance increases as well as, whereas the curve of Graph stays close to the ideal curve. Finally, Graph achieves a 1.25 speedup on 2048 in comparison to Synchronous.

The third set of experiments explores Tarragon's ability to tolerate increasingly higher communication latencies. In these experiments latencies are artificially increased by increasing the amount of data sent when exchanging ghost cells. Figure 2 illustrates a comparison between Synchronous and Graph solving a 1000^3 problem on 64 cores. It can be observed how the timings of Synchronous (blue line) exhibits the same behavior of the communication cost (red line) whereas the timing of Graph (green line) increases gracefully and at a much lower rate.

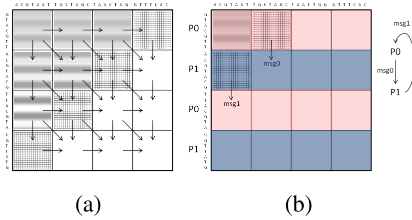
3.2 Needleman-Wunsch

In computational molecular biology, comparing sequences (e.g. DNA) is a fundamental primitive operation and it is at the basis of more complex manipulations and analysis [4]. The comparison between two sequences can provide a measure of how similar the sequences are, for example by quantifying the minimum number of transformations to make the sequences equal. Similarly, it is useful to *stretch* the sequences by inserting special characters in order to maximize the number of matching characters. Tools that solve this type of problems are used in DNA sequences manipulations, such as fragments assembly, and analysis, such as database searches based on similarity.

Needleman-Wunsch is based on the computation of the *edit matrix* for the given input sequences, which are usually referred to as *reference* and *query*. Rows and columns are associated to the characters of the query and the reference, respectively, and each entry of the edit matrix is the score of the alignment between the corresponding prefixes of the sequences. At the end of the computation, the bottom-right corner holds the score of the complete alignment.

Scores are assigned according to a *scoring function* $\Sigma \times \Sigma :\rightarrow \mathbb{Z}$ and a recurrence relation. The scoring function assigns scores to matches and mismatches whereas the penalty associated with the insertion of a *gap* (_), where a gap represents simple genetic mutations causing insertion or deletion of a base, is accounted for by the recurrence relation. The recurrence relation defines how scores are assigned when prefixes are extended. Implicitly, the recurrence relation defines the dependencies between entries.

Needleman-Wunsch has two phases: the first phase computes the *edit matrix*; the second phase traverses the edit matrix along the path that corresponds to the optimal alignment. The computational complexity of the first phase is $O(mn)$, where m and n are the length



(a) (b)

Figure 3: Needleman-Wunsch alignment. Figure 3a shows the wavefront execution on a matrix decomposed into 16 blocks, the arrows represent the dependencies between blocks. Striped block have been filled already; tiled blocks are ready for execution and can execute concurrently, that is the wavefront; white blocks cannot be computed yet. Figure 3b shows the wavefront after the completion of the second diagonal and the mapping to two processes. At this time the processes form a communication ring. The ring is represented on the right of the matrix.

of the two sequences, assuming that filling each entry of the matrix has constant cost. The assumption holds for the class of scoring functions considered in this dissertation. Finding the path has linear complexity because it traverses the matrix from the bottom-right corner to the top-left corner, and each step is either a move up or a move to the left and has constant cost. It follows that the first phase accounts for the majority of the computation time. For this reason, only the first phase is considered.

The parallel algorithm here considered is a wave-front algorithm, as illustrated in Figure 3. Four implementations are compared in the remainder of this section: a Synchronous MPI implementation, an Asynchronous MPI implementation, and two Tarragon implementations. The different codes are referred to as Synchronous, Asynchronous, Block, and Panel. The description of each implementation follows.

In Synchronous, blocks are assigned to processes using a 1- dimensional cyclic mapping along the columns and each process computes the blocks owned, row by row, left to right. Communication takes place after each process completes a block. After completing a block, processes send the bottom of the computed block forming a communication ring (Fig. 3b). The ring is implemented using blocking communication primitives and the exchange is done in two phases: first even-rank processes send and odd-rank processes receive, then even-rank processes receive and odd-rank processes send.

Asynchronous implements the communication phase differently. In Asynchronous processes maintain a communication schedule and post non-blocking receives in advance. Then, as the computation advances, processes follow the communication schedule retiring completed receives and posting new ones. To this end, Asynchronous tracks the window of pending receives using a circular queue. Similarly, sends are posted as soon as possible but retired only later, as the window of pending sends slides forward.

In Block each block is associated with a task. Tasks execute after receiving a message from the task on the left, and the task from the top. The two messages provide data on the leftmost column of the block and the first row, and therefore provide all the necessary data to compute the associated block. When executing, the task fills the associated block of the edit matrix. Finally, the task terminates by sending the last row and the rightmost column to the task below and the task on the right. In fact, the graph can be obtained by the blocked representation of the matrix and the corresponding dependencies, with the exception that there is no diagonal edge. The diagonal dependency is implied by the vertical edge.

In Panel a whole block row is associated with a task. Because each block in a row can execute only after the preceding block, the amount of parallelism is not reduced. The computation is identically pipelined. However, the size of the graph is reduced and so the memory utilized to store the graph. In addition, the RTS has to keep track of a much smaller set of tasks and can therefore be more responsive in handling communication events. Within a task the computation and the communication granularity is not different. Task compute a block at a time and as soon as a block is completed, data is sent on the outgoing edge. In this way the granularity of the tasks is coarsened, while the granularity of the computation and, very importantly, the granularity of the phases of the pipeline are not.

3.2.1 Performance Evaluation

The first experiments on the Intel cluster compare the four implementation on 2 and on 8 cores. In addition, Block and Panel are tested in two different configurations. The two configurations are single threaded and multi threaded. The relevance of the single-thread version is that since the computation is CPU bound, assigning one core to the service thread might cause a significant performance loss. It is therefore important to assess the performance loss and identify the optimal node configuration for small scale runs. By doing so it is also possible to observe whether the optimal configuration changes at large scale.

Length	Cores	Synch	Asynch	TB1	TP1	TPM
2^{17}	8	10.9	10.8	10.0	10.3	11.4
2^{18}	32	11.5	11.2	10.2	10.6	11.8*
2^{19}	128	11.9	11.5	10.6	11.3	12.5*
2^{20}	512	15.6	13.2	12.2*	12.6	13.7*
2^{21}	2048	14.3	14.6	15.2*	15.9*	14.4*

Table 3: Running times on the Intel cluster (Abe@NCSA). TB1 is Block running single threaded. TP1 is Panel running single threaded. TPM is Panel running multi threaded. Timings are given in seconds. The table presents weak-scaling run times in comparing two sequences of equal length, as reported in the first column.

Scaling up to 512 cores, the single-thread configurations are still faster than the MPI implementations. However, on 2048 cores the cost of managing asynchronous communication requests suddenly increases and affects all the codes using non-blocking communication primitives, especially the single-thread configurations. The same is true also for Asynchronous and in fact, Synchronous is the fastest indicating that a performance bottleneck in the underlying communication layer is affecting the non-blocking primitives. Panel running multi threaded achieves similar performance (less than 0.7% difference in running time) showing that as the core count increases, and so does the communication cost, the multi-thread configuration becomes viable again.

Finally, the computation proceeds in a pipelined fashion. Each core executes one stage of the pipeline. In order to minimize idle times

the pipeline must be kept full at all times. In the MPI implementations the pipeline is implicitly defined as part of the hard coded schedule. The schedule is optimal and executes first the tasks with highest number of descendants in the graph. In Tarragon tasks are scheduled dynamically without any predefined order. To ensure that progress is made while maintaining a pool of available tasks, tasks are prioritized giving higher priority to the tasks closer to the top of the matrix. On the other hand tasks with low priority can execute when all the high priority tasks are blocked. The result is that Tarragon is able to follow a theoretically optimal schedule expressed by priorities and to adapt to communication delays by executing low priority tasks instead of waiting.

Table 4 shows the difference between the implementations with and without prioritized execution. In some cases prioritized execution is crucial to achieve high performance. Especially for Panel, where the lower number of tasks reduces the flexibility in scheduling, prioritized execution prevents lower priority panels to execute for a long time and to stall the pipeline. The negative effect of stalling is even more evident as the number of cores increases. The more cores there are, the longer it takes to completely fill the pipeline after stalling. The result is that prioritized execution accounts for a reduction in run time of approximately 20% on 512 and 2048 cores. Single threaded Panel is the least affected because it has less opportunities to alternate between execution and scheduling and naturally follows the order in which tasks are enabled. The order corresponds to the prioritized order since panels are enabled in descending order, starting from the first row.

3.3 Sparse LU Factorization

LU factorization is employed in direct solvers of systems of linear equations. In such solvers, given a linear system of the form $Ax = b$, first A is factorized, via some variant of Gaussian Elimination, into two matrices which are a lower and an upper triangular matrix (L and U); then, the resulting triangular systems, $Ly = b$ and $Ux = y$, are solved by means of forward and backward substitution, respectively. The result of the latter solve gives the solution to the system $Ax = b$.

To take advantage of cache locality and of highly-tuned single-core dense linear algebra libraries, factorization algorithms are implemented using blocking. With blocking, the algorithm is expressed in terms of submatrices of contiguous rows and columns rather than individual rows and columns, and operations between submatrices are executed by invoking dense linear algebra routines. In parallel formulations, blocking also results in more efficient communication because it increases messages length while decreasing their number than for un-blocked algorithms. Consequently, the available bandwidth is used more efficiently and the total communication cost is reduced.

As reference MPI implementation, we used the SuperLU_DIST software package [28]. The solver in SuperLU_DIST is a distributed memory sparse direct solver for general systems of equations[3]. The solver is characterized by a symbolic factorization phase in support of a static pivoting strategy [18]; during the symbolic factorization the matrix is transformed to ensure stability and to preserve sparsity. SuperLU_DIST also employs software pipelining to overlap communication with computation.

[3]Systems of equations characterized by a positive-definite matrix can be solved more efficiently using Cholesky factorization.

The Tarragon implementation of factorization is embedded in SuperLU_DIST and replaces the original factorization based on MPI. The Tarragon implementation effects a compromise between creating a large number of small tasks and maintaining the coarse granularity of the original formulation. Creating a large number of small tasks would be ideal for exposing a high degree of parallelism. However, preserving interoperability with other SuperLU_DIST routines introduces limitations on the design choices in the Tarragon implementation. For example, due to permutations performed in the symbolic factorization phase, rows of L are stored out of order, and the order differs within each block column. In addition, blocks of U are stored as columns of different size. As a result, factorization involves searches and data *formatting* operations before dense linear algebra routines can be used. Such operations would be duplicated in a fine grained task-parallel implementation causing high computational overheads. The approach adopted in the Tarragon factorization algorithm is conservative in that it is task-parallel, but the tasks defined match the granularity of the phases of the MPI implementation.

There are also limitations on the way factorization can be executed in the Tarragon implementation. The SuperLU_DIST design assumes an underlying SPMD execution model (MPI) and it uses a block-cyclic mapping of blocks to processes. With Tarragon, when a single multi-threaded run-time system is deployed on each node, a block-cyclic distribution maps consecutive blocks on different nodes causing a significant increase in inter-node communication. For this reason, the Tarragon implementation runs more efficiently if it follows the same process mapping of SuperLU_DIST, though this prevents the run-time system from transferring data via shared-memory and the available parallelism is limited. In addition, while the data structures used in SuperLU_DIST are stored in buffers ready for communication with MPI, in Tarragon such data structures are also be integrated with a message header requiring extra memory copies.

3.3.1 Performance Evaluation

The test suite used for the experiments is composed of twelve matrices from real world science and engineering problems. Eight matrices are taken from the University of Florida Sparse Matrix Collection [10, 26], two from a fusion energy study [1], one from an accelerator structural design problem [2], and one is a dense matrix. The matrices were selected with different sizes, number of nonzeros, sparsity, and sparsity pattern to ensure that performance is evaluated under different conditions. In particular, the dense matrix generated for this study is used to create a balanced workload distribution in the attempt to isolate communication delays due to data transfer from delays due to load imbalance.

For most of the small matrices, the peak performance (timing in boldface) is achieved on a small number of cores due to load imbalance and the little available parallelism. However, in many cases the Tarragon implementation achieves its peak on a larger number of cores. However, the Tarragon implementation is less efficient in some cases in which, on an equal number of nodes, it is slower than the SuperLU_DIST implementation. Overall, on the small matrices, the Tarragon implementation achieves a 1.09 average speedup over the SuperLU_DIST implementation.

On two of the large matrices, the Tarragon implementation outperforms the SuperLU_DIST implementation. However, on matrix *matrix181*, the Tarragon implementation does not improve its performance when scaling from 256 cores to 512 cores, whereas the

Cores	TB1	TB1+	sp(%)	TP1	TP1+	sp(%)	TPM	TPM+	sp(%)
8	10.0	10.0	0	10.3	10.3	0	11.4	11.4	0
32	10.2	10.2	0	10.6	10.6	0	12.1	11.8	3
128	10.6	10.6	0	11.3	11.3	0	13.3	12.5	6
512	13.0	12.2	7	12.6	12.6	0	17.3	13.7	26
2048	17.1	15.2	13	16.3	15.9	3	17.6	14.4	22

Table 4: Running times on the Intel cluster (Abe@NCSA). TB1 is Block running single threaded. TP1 is Panel running single threaded. TPM is Panel running multi threaded. Timings are given in seconds. The table presents weak-scaling run times in comparing two sequences of equal length, as reported in the first column.

Table 5: Comparison of peak performance on Abe. The table compares the peak performance that the two implementations achieve in the LU factorization, and gives the speedup achieved with Tarragon (column *sp*).

Matrix	SuperLU		Tarragon		Sp
	GFLOPS	Cores	GFLOPS	Cores	
bbmat	12.6	16	14.4	16	1.14
g7jac200	12.2	16	14.3	32	1.17
inv-extr-1	7.7	16	8.6	16	1.10
matrix31	11.3	8	12.7	16	1.13
mixing-tank	15.2	16	17.7	16	1.17
nasasrb	8.6	8	8.7	16	1.01
stomach	8.5	8	9.5	16	1.12
torso1	14.1	8	13.4	16	0.95
twotone	2.8	16	2.9	16	1.05
dense	203.2	512	235.4	512	1.16
dds15	19.0	64	19.5	64	1.03
matrix181	85.9	512	82.1	256	0.96

SuperLU_DIST implementation, which is significantly slower on 256 cores, enjoys a performance improvement scaling to 512 cores. As a result, the SuperLU_DIST implementation is faster than the Tarragon implementation. Overall, on the large matrices, the Tarragon implementation achieves a 1.05 average speedup over the SuperLU_DIST implementation.

While Tarragon achieves overlap and meets the performance of the SuperLU_DIST implementation, which is also overlapping communication with computation, the potential of applying Tarragon to sparse LU factorization is not completely expressed. In particular, the data decomposition and mapping imposed on Tarragon limits its ability to execute one process per shared memory node and to execute fine-grained tasks. A finer decomposition would create more opportunities for scheduling tasks adapting to communication delays and achieving better overlap. Future research should explore ways to enable the conditions that are most favorable to Tarragon, starting from the symbolic factorization.

4. CONCLUSIONS

Tarragon is a programming model that supports latency-hiding applications in scientific computing. Tarragon introduces a novel task graph abstraction that enables the programmer to express parallelism and data dependencies, and that shields the programmer from the complexity of communication, threading, and scheduling details. The task graph and its attributes create a separation of structural and correctness concerns from performance concerns. In addition, by virtue of its data-driven execution model, Tarragon automatically overlaps communication with computation.

The results in the application studies demonstrate Tarragon's ability to achieve high performance and to realize overlap. In all of the applications, the Tarragon implementation meets or exceeds the performance of the corresponding overlapping MPI reference most of the times. In addition, even when split-phase coding affects the memory access pattern and negates the performance improvement due to overlap, Tarragon supports overlapping algorithms that preserve locality. For example, results with the Finite-Difference solver show that the MPI overlapping version suffers a performance loss compared to the non-overlapping MPI version, whereas the Tarragon version exceeds the performance of the non-overlapping MPI version owing to overlap and good locality.

The results of the sequence alignment application demonstrate that the Tarragon version achieves overlap and that its performance is maximized by an appropriate scheduling policy. While Tarragon always achieves overlap, default scheduling policies might cause execution to diverge from the critical path. However, with graph analysis and prioritization users can identify and establish the most appropriate scheduling policies. These results demonstrate that graph-metadata can be used to tune performance without affecting correctness of execution.

In conclusion, this work demonstrates that data-driven execution coupled with metadata abstractions support latency tolerance. In addition, Tarragon's programming model supports performance optimization techniques that are decoupled from the algorithmic formulation and the control flow of the application code; while enabling performance tuning, the resulting separation of concerns between performance and correctness promotes performance portability.

4.1 Future Research Directions

The explicit task graph representation used in Tarragon lends itself to analysis-driven performance optimizations. For example, graph analysis can suggest a task redistribution that minimizes inter-node communication and hence the overall communication cost. Using a richer set of metadata, perhaps collected partly by the application and partly by the run-time system, offers promise for even more complex optimizations. Load balancing is such an example; if the graph is annotated by the run-time system with *measured* load information, or by the application with *expected* load information, graph analysis can determine the optimal workload distribution.

Tarragon can be extended to support clusters of accelerators (i.e. GPUs), an area in which communication latencies are a major performance bottleneck. The idea of supporting hybrid code that executes on clusters of multicore nodes with accelerators appears to be promising in a context in which metadata can support dynamically tuned scheduling. Future research should also focus on graph analysis and metadata annotation as a way for defining scheduling

policies to optimize execution on clusters of accelerators.

Tarragon mostly delegates productivity concerns to library extensions. However, the use of libraries in refactoring and porting existing code bases to emerging architectures still requires some degree of redevelopment. A promising approach in refactoring existing code is to use automatic translation tools [22]. To this end, automatic translation to apply transformations and optimize MPI code is an active area of research, including an effort for translating MPI code to Tarragon code.

Acknowledgment

This research was supported in part by the NSF contract ACI0326013; in part by a grant from the NSF's Office of Cyberinfrastructure entitled "Integrated Performance Monitor"; and in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. It used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231, and resources of Teragrid provided by NCSA, TACC, ORNL, and SDSC.

5. REFERENCES

[1] Center for Extended MHD Modeling (CEMM). `http://w3.pppl.gov/cemm`.

[2] Community Petascale Project for Accelerator Science and Simulation (COMPASS). `https://compass.fnal.gov/`.

[3] A. Krishnamurthy and D. E. Culler and A. Dusseau and S. C. Goldstein and S. Lumetta and T. von Eicken and K. Yelick. Parallel programming in Split-C. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 1993. ACM Press.

[4] Aluru, S., editor. *Handbook of Computational Molecular Biology*, chapter 1. Computer and Information Science. Chapman & Hall/CRC, 2005.

[5] Anthony Danalis and Ki-Yong Kim and Lori Pollock and Martin Swany. Transformations to Parallel Codes for Communication-Computation Overlap. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2005. IEEE Computer Society.

[6] Arvind, K. and Nikhil, Rishiyur S. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, 39, March 1990.

[7] Asanovic, Krste et. al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[8] Colella, Phil. Designing Software Requirements for Scientific Computing, 2004.

[9] Costin Iancu and Parry Husbands and Paul Hargrove. HUNTing the Overlap. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Duff, I. S. and Grimes, Roger G. and Lewis, John G. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1), 1989.

[11] Husbands, Parry and Yelick, Katherine. Multi-threading and one-sided communication in parallel LU factorization. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, New York, NY, USA, 2007. ACM.

[12] J. R Gurd and C. C Kirkham and I. Watson. The Manchester prototype dataflow computer. *Commun. ACM*, 28(1), 1985.

[13] Jack B. Dennis. Data Flow Supercomputers. *Computer*, 13(11), Nov 1980.

[14] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture*, New York, NY, USA, 1975. ACM Press.

[15] Kalé, L.V. and Krishnan, S. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In Paepcke, A., editor, *Proceedings of OOPSLA'93*. ACM Press, September 1993.

[16] Laxmikant V. Kalé. The Virtualization Model of Parallel Programming : Runtime Optimizations and the State of Art. In *LACSI 2002*, Albuquerque, October 2002.

[17] Leslie G. Valiant. A Bridging Model for Parallel Computation. *CACM*, 33(8), Aug 1990.

[18] Li, Xiaoye S. and Demmel, James W. Making sparse Gaussian elimination scalable by static pivoting. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, Washington, DC, USA, 1998. IEEE Computer Society.

[19] Patterson, David A. Latency lags bandwith. *Commun. ACM*, 47, October 2004.

[20] Pietro Cicotti and Scott B. Baden. Short Paper: Asynchronous programming with Tarragon. *High Performance Distributed Computing, 2006 15th IEEE International Symposium on*, 2006.

[21] Planas, Judit and Badia, Rosa M. and Ayguadé, Eduard and Labarta, Jesus. Hierarchical Task-Based Programming With StarSs. *Int. J. High Perform. Comput. Appl.*, 23, August 2009.

[22] Quinlan, Daniel J. and Miller, Brian and Philip, Bobby and Schordan, Markus. Treating a User-Defined Parallel Library as a Domain-Specific Language. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, Washington, DC, USA, 2002. IEEE Computer Society.

[23] R. D. Blumofe and C. F. Joerg and B. C. Kuszmaul and C. E. Leiserson and K. H Randall and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[24] Scott B. Baden and Stephen J. Fink. Communication overlap in multi-tier parallel algorithms. In *Proc. of SC '98*, Orlando, Florida, November 1998.

[25] Thorsten von Eicken and David E. Culler and Seth Copen Goldstein and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, New York, NY, USA, 1992. ACM Press.

[26] Timothy A. Davis. university of Florida Sparse Matrix Collection. 1994.

[27] Top 500. http://top500.org.

[28] Xiaoye S. Li and James W. Demmel. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Mathematical Software*, 29(2), June 2003.