DaSH: A Benchmark Suite for Hybrid Dataflow and Shared Memory Programming Models

With Comparative Evaluation of Three Hybrid Dataflow Models

Vladimir Gajinov*† Srđan Stipić*† Igor Erić$^\Psi$
Osman S. Unsal* Eduard Ayguadé*† Adrián Cristal§‡

* Barcelona Supercomputing Center † Univesitat Politecnica de Catalunya ψ University of Belgrade
§ Artificial Intelligence Research Institute ‡ Spanish National Research Council

vladimir.gajinov@bsc.es srdjan.stipic@bsc.es igor_eric21@yahoo.co.uk
osman.unsal@bsc.es eduard.ayguade@bsc.es adrian.cristal@bsc.es

Abstract—**The current trend in development of parallel programming models is to combine different well established models into a single programming model in order to support efficient implementation of a wide range of real world applications. The dataflow model has particularly managed to recapture the interest of the research community due to its ability to express parallelism efficiently. Thus, a number of recently proposed hybrid parallel programming models combine dataflow and traditional shared memory. Their findings have influenced the introduction of task dependency in the recently published OpenMP 4.0 standard.**

In this paper, we present DaSH - the first comprehensive benchmark suite for hybrid dataflow and shared memory programming models. DaSH features 11 benchmarks, each representing one of the Berkeley dwarfs that capture patterns of communication and computation common to a wide range of emerging applications. We also include sequential and shared-memory implementations based on OpenMP and TBB to facilitate easy comparison between hybrid dataflow implementations and traditional shared memory implementations based on work-sharing and/or tasks. Finally, we use DaSH to evaluate three different hybrid dataflow models, identify their advantages and shortcomings, and motivate further research on their characteristics.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *frameworks, modules, packages, patterns.*

General Terms
Measurement, Performance, Design, Experimentation.

Keywords
Dataflow, shared memory, transactional memory.

1. INTRODUCTION

Shared memory multicore processors are the current workhorses of the computing industry. A variant of Moore's Law states that the number of cores on chip will double every two years. However, one of the major issues with this architecture is the cost of synchronization that often prevents shared-memory programs from scaling to high core counts. Solutions to this problem usually require significant programmer effort. This has caused a renewed interest into dataflow due to its ability to express parallelism efficiently.

The main characteristic of the dataflow model [2,6] is that the execution of an operation is constrained only by the availability of its input data. Following a single assignment rule, the dataflow model is able to extract all the parallelism inherent in a program. Yet, there are many applications, such as a reservation system or chess, for which the state is a fundamental part of the problem. Expressing these problems in a dataflow model is possible, but is often inefficient.

Due to the diversity of algorithms and applications, it is therefore unlikely that a single programming model can support efficient implementation of the whole application spectrum [3]. Thus, a number of recent proposals aim to combine different well established models into a single programming model that can support a wider range of computation and communication patterns. OmpSs [19], Atomic Dataflow (ADF) [8] and Intel Threading Building Blocks (TBB) [13] are some of the models that have recognized the potential of extending the shared memory model with support for dataflow. While each of these models uses its custom set of applications to prove the concept and demonstrate its potential, there is a necessity for an extensive benchmarks suite that will feature not only applications that lend themselves well to dataflow, but also those for which some other parallel programming paradigm, such as work-sharing or tasking, may be more suitable. To the best of our knowledge, no such benchmark suite exists at the moment.

The main contribution of this paper is DaSH – the first comprehensive benchmark suite for hybrid dataflow and shared memory programming models[1]. While developing DaSH, we have followed the approach from Berkeley that has identified a set of 13 dwarfs that capture patterns of communication and computation common to a wide range of emerging applications [3]. Accordingly, each DaSH benchmark represents an application from a single Berkeley dwarf. Currently, DaSH features 11 benchmarks. For each benchmark, we provide sequential implementation, two shared memory implementations based on work-sharing and/or tasking (one that uses OpenMP and the other that uses TBB) and three dataflow implementations realized using the OmpSs, ADF and Intel TBB Flow Graph hybrid dataflow models.

Our second contribution is the evaluation of these three hybrid dataflow models that demonstrates the usefulness of DaSH for the research of these models. In particular, we show that:

CF'14, May 20 - 22 2014, Cagliari, Italy
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2870-8/14/05…$15.00
http://dx.doi.org/10.1145/2597917.2597942

[1] The DaSH benchmark suite is available at https://www.bscmsrc.eu/software/dash-benchmark-suite

- Hybrid dataflow models support a straightforward implementation of certain types of irregular algorithms by allowing the developer to naturally represent the algorithm, which results in better programmability and/or performance. For example, the dataflow implementation of the sparse LU factorization performs 27% better than the corresponding shared memory implementations.
- The main strength of these models is the ability to eliminate unnecessary barriers and thus expose more parallelism. Most of the performance gain when using these models is a result of the increased parallelism and decreased thread idle time. In addition, we have identified applications for which dataflow provides barrier-free implementation even when the algorithm inherently depends on barriers.
- There are important differences between the three hybrid dataflow models that we evaluate, both in terms of programmability and performance. We identify some of their shortcomings and suggest possible improvements.

The rest of the paper is organized as follows. In Section 2 we give an overview of hybrid dataflow models. In Section 3 we present the DaSH benchmark suite. Next, we provide a study of hybrid dataflow models, namely OmpSs, ADF and TBB Flow Graph, comparing them with traditional shared memory models, namely OpenMP and TBB. In Section 4 we analyse the programmability of the DaSH benchmarks and then in Section 5 we evaluate the performance of corresponding implementations. Section 6 summarizes the related work. Finally, in Section 7 we conclude and discuss the future work.

2. HYBRID DATAFLOW MODELS

This section provides an overview of hybrid parallel programming models that combine dataflow and shared memory programming. Models such as OmpSs [19], ADF [8] and TBB Flow Graph framework [13] are based on execution of tasks that are scheduled, according to the dataflow principles, when their input dependencies are satisfied. Typically, a programmer explicitly defines input and/or output dependencies for each task. This information is then used by the runtime system to construct task dependency graph that governs the execution of a program. As tasks execute, they produce data on which other program tasks depend on. Once all input dependencies for a given task are satisfied, the task becomes enabled and can be scheduled for execution. The tasks are executed by worker threads and scheduling is typically based on work-stealing.

Contrary to a pure dataflow model, which assumes side-effect free execution of dataflow tasks, a hybrid dataflow model can benefit from the underlying shared memory architecture by allowing dataflow tasks to share data. Particularly, in a pure dataflow model, each time the data is updated a new copy is produced. Inherently, the problem occurs when the updated data is some complex structure, such as an array, which effectively makes updates expensive. Hybrid dataflow model can avoid this problem by treating such complex data as a shared state and updating it in place. Naturally, accessing shared state may require synchronization. In this work we employ transactional memory (TM) [11] for the shared state synchronization because it integrates seamlessly into the dataflow model. In particular, both abstractions exhibit isolation property: dataflow in terms of execution of data dependent tasks and transactional memory in terms of concurrent accesses to the shared state by different sharers.

In this work, we are use OmpSs [19], the ADF model [8] and the Flow Graph extension of the Intel TBB [13] to implement the DaSH benchmark suite and establish its practicality for the evaluation of hybrid dataflow models. Hence, we continue this section with an overview of these three models.

2.1. OmpSs

OmpSs extends OpenMP with a support for asynchronous parallelism that is based on execution of data-dependent tasks. Specifically, OpenMP task directive is extended with three additional clauses - *in*, *out* and *inout* - that a programmer may use to explicitly declare data dependencies for a task. When a new task is created, the runtime system matches its *in* and *out* dependencies against dependencies of all existing tasks. If the match is found, the new task becomes a successor of the corresponding tasks. This process creates a task dependency graph at runtime. Tasks are scheduled for execution as soon as all off their predecessor in the graph have finished, or at creation, if they have no predecessors. The task construct is further extended with the *concurrent* clause that allows a number of instances of the task to execute simultaneously. However, a programmer must ensure that additional synchronization is used inside a task, if necessary.

2.2. Intel TBB Flow Graph

Starting from version 4.0, Intel Threading Building Blocks (TBB) framework [13] includes Flow Graph extension that enables construction of task dependency graphs. The flow graph consists of three primary components: a graph object, nodes, and edges. The graph object provides methods to run the graph and to wait for its completion. Nodes generate, buffer, or transform messages and data. In particular, functional nodes execute user code. Buffering nodes implement different forms of data buffering. The rest of the nodes support various forms of control and communication, such as split, join, broadcast etc. Concurrency of functional nodes can be explicitly controlled, which means that a number of instances of the same node can be executed in parallel if multiple data items exist on the node's input port. Edges connect the nodes and represent channels through which nodes communicate and exchange data. In a nutshell, programmers explicitly create nodes and edges that express computations and dependencies between these computations.

2.3. Atomic Dataflow Model

The basic building blocks in the ADF model are ADF tasks that are defined using the *adf_task* pragma directive with a set of clauses. The *trigger_set* clause defines input dependencies for a task that are used by the runtime system to construct a task dependency graph that governs the execution of a program. The *instances* clause instructs the runtime to create a number of instances of the same task. The *until* clause specifies the exit condition of the task. The *execute* clause terminates the task after a given number of task invocations. The *pin* clause is used to bind a task to a given thread. Finally, the *relaxed* clause instructs the runtime to switch off the implicit TM synchronization of the task body. Conceptually, an ADF task operates as a macro dataflow actor that processes one set of data, produces output data and then waits for a new matching set of input data. The main thread uses the *adf_start* and the *adf_taskwait* pragma directives to initiate dataflow execution and to wait for its completion.

2.4. The Differences between Models

The main difference between these three models is the type of the dependency graph that the model relies on: OmpSs constructs a task dependency graph, while the other two models build a data dependency graph. In the next section, we will show how this choice may affect the programmability of a given model. Further,

```
SomeType Produce();
void Consume(SomeType);
SomeType consumeToken;

void main() {
    while ( itemsToProduce-- ) {
        #pragma omp task out(consumeToken)
                         concurrency(nthreads)
        SomeType item = Produce();

        #pragma omp task in(consumeToken)
                         concurrency(nthreads)
        Consume(item);
    }

    #pragma omp taskwait
}
```

a)

```
void main() {
    #pragma adf task instances(nthreads)
                     execute(numItemsToProduce)
    consumeToken = Produce();

    #pragma adf task trigger_set(consumeToken)
                     instances(nthreads)
    Consume(consumeToken);

    #pragma adf start
    #pragma adf taskwait
}
```

b)

```
void main() {
    graph g;

    function_node<continue_msg, SomeType> producer
       (g, concurrency=nthreads, [=] () -> SomeType {
            return Produce();
       });

    function_node<SomeType> consumer
       (g, concurrency=nthreads, [=](SomeType input)->void {
            Consume(input);
       });

    make_edge(producer, consumer)

    for (int i = 0; i < numItemsToProduce; i++)
       producer.try_put(continue_msg());

    g.wait_for_all();
}
```

c)

Figure 1: Multiple Producer - Multiple Consumer implementation in: a) OmpSs, b) the ADF model, and c) the TBB Flow Graph framework.

OmpSs builds a graph dynamically while the tasks are being created, whereas the ADF model and the TBB Flow Graph framework build their graphs statically, before the start of the dataflow execution. However, in these two models, tasks are being reused as many times as their input dependencies are satisfied. On the contrary, in OmpSs each task executes only once, after which it is discarded, thus maintaining similar semantics with OpenMP tasks.

The models also differ in a programming interface through which they allow a user to express dependencies. Figure 1 illustrates the use of dataflow constructs of these three models for the implementation of multiple producer – multiple consumer problem. OmpSs uses task directive data directionality clauses to instruct the runtime how to automatically connect tasks in existing task graph - Figure 1a. In the ADF model, a programmer expresses dependencies using dataflow tokens - Figure 1b. These are special variables that carry data between graph nodes (tasks). The runtime system uses input task dependency information, defined using the *trigger_set* clause with a list of tokens, to construct the graph automatically. This relieves a programmer from the burden of explicitly connecting all the nodes. On the contrary, using TBB, a programmer has to construct entire graph explicitly by connecting the output port of a given node with an input port of a consumer node - Figure 1c. The TBB framework provides *make_edge* template function for this purpose. Thus, both the nodes and the edges must be defined explicitly.

3. THE DASH BENCHMARK SUITE

Experts from various computing domains have previously identified 13 Berkeley dwarfs [3] that capture essential communication and computation patterns found across a wide range of emerging applications. Since *breadth* is one of the most important characteristics of any benchmarks suite, our decision was to develop DaSH by implementing a single application from each dwarf category. Currently, DaSH consists of 11 benchmarks. Two dwarfs are not supported. Combinatorial logic dwarf describes problems that exploit bit-level parallelism by performing simple operations on very large amounts of data (computing checksums or CRCs). Since the execution is heavily dominated by the time to read the data, the benefits of parallelization are quite limited and so is the usefulness of this dwarf for programming model evaluation. We are currently working on the last missing dwarf, *Graph traversal*, that we plan to include in DaSH in a near future. Next, we describe characteristics of each DaSH benchmark.

Branch-and-Bound – This dwarf represents a general type of algorithms for finding optimal solutions of various global optimization problems. The DaSH benchmark for this dwarf implements a branch-and-bound algorithm to solve the Traveling Salesman Problem (TSP) by solving equivalent assignment problem (AP) that provides lower bounds for the solution of the TSP problem. The algorithm is based on two lists: one with solved assignment problems, and the other with newly generated problems derived from branching previously solved APs. The algorithm iterates through the problem list, prunes suboptimal problems and solves the rest until the list is empty. A moderate synchronization is necessary to protect list operations and currently found optimal solution.

Shared memory implementations first create a dedicated task for each problem from the current problem list. When the worker threads finish executing all these tasks they reach the task barrier. Next, a single thread removes the first problem from the list of solved problems and uses it to generate new subproblems that may lead to an optimal solution, and then restarts the parallel section. This continues until both problem lists are empty.

The OmpSs implementation is similar to the OpenMP implementation because this is the most natural approach to the problem using this model. The ADF and TBB Flow implementations try to avoid barriers by relying on a *Generator* task that is enabled each time a problem is solved by some *Solve* task. The *Generator* task takes the problems from the start of the solved problem list, generates new subproblems and stores them into the problem list, and then creates a number of output tokens that carry pointers to the most promising problems from the problem list. These tokens enable further executions of the *Solve* tasks. The limit on how many tokens will be generated by the *Generator* task can be controlled by a command line parameter. However, as this parameter gets larger the algorithm does more branching then bounding, which leads to a greedy solution.

Dense linear algebra - This dwarf represents the classic dense vector and matrix operations. We demonstrate this dwarf with the block Cholesky decomposition. The algorithm decomposes positive-definite matrix *A* into the product of a lower triangular matrix and its conjugate transpose $A = LL^T$. We describe parallel implementations of this algorithm in Section 4.1.1.

Dynamic programming – Dynamic programming is used in a variety of problems, such as technology mapping in integrated circuit design, longest common subsequence matching of DNA strands and various optimization problems. The essence of this dwarf is that the solution of a problem is based on solving simpler overlapping subproblems. Data dependencies are created between levels of subproblems since an optimal solution to a larger problem depends on an optimal solution to its subproblems. Therefore, these algorithms can be solved naturally using dataflow.

Often, subproblems can be grouped into blocks to increase computational granularity, which is a method that we apply in a solution of the unbounded Knapsack problem that represents this dwarf in DaSH. The formulation of the problem is the following: given a set of items, each with a weight and a value, determine the number of each item to include in a knapsack so that the total weight is less than or equal to the knapsack capacity, maximizing the total value. The solution is based on a two dimensional matrix *M* with *n* rows that represent the items and *C* columns that represent the knapsack capacities [9]. The algorithm goes through matrix *M* row by row. For every item *t,* the maximum number of items of type *t* that can be loaded given a knapsack with capacity *cap* is computed as $\lfloor cap/weight[t] \rfloor$ A choice for the number of units of product *t* to load in the knapsack is notated as *u*. Given a choice of *u* units for product *t*, a possible value for entry *(t, cap)* is calculated adding the previously found maximum value in row *t-1* and column *cap - (weight[t]∗u)* to the value from loading *u* units of item *t*, that is *u∗value[t]*. After repeating this computation for all possible values of *u, (u = 0* ..$\lfloor cap/weight[t] \rfloor$ *)*, the maximum of these values is stored in entry *(t, cap).*

Shared memory implementations are based on parallel processing of a single row at the time, after which there is an implicit barrier that separates the processing of successive rows. Dataflow implementations avoid barriers by partitioning each row to *n* partitions and creating *n* tasks that process these partitions, where *n* is the number of threads in execution. Dependency structure is such that each partition *p[i,j]* can be processed only after partitions *p[i-1, j]* and *p[i, j-1]* have been processed, which effectively resembles a wavefront processing of row partitions.

Finite state machine - This dwarf represents a system whose behaviour is defined by states and transitions between states based on given inputs and events. Parallelism is often difficult to utilize. However, some state machines can be decomposed into multiple simultaneously active state machines. One such example is a deterministic finite state machine for the text pattern search, which is the problem solved by the DaSH benchmark for this dwarf. The states are integer values from *0* to *N*, where *N* is the length of the pattern. We build state transition matrix where each character has one of the *N* possible states based on the search pattern. This transition matrix is used during the text traversal to change the current state of pattern matching machine. When the machine reaches the final state, a new pattern match has been found and we record its index in the text.

Shared memory solutions are implemented using tasks because partition boundaries have to be properly adjusted to assure that the words are not split. Each task searches a part of the content string. Synchronization is necessary when a thread has to update the global list of occurrence indexes. Dataflow solutions are reduced to fork-join parallelism since each partition can be processed independently from other partitions.

Graphical models - This dwarf describes a group of algorithms based on a graph in which nodes represent variables and edges represent conditional probabilities (dependencies). Examples include Bayesian networks, Hidden Markov Models and neural networks. These applications typically involve many levels of indirection, and a relatively small amount of computation. The DaSH benchmark for this dwarf implements the Viterbi algorithm for the Hidden Markov model [20]. The goal of the algorithm is to find the most likely sequence of states that can explain given observations. The algorithm iterates through a sequence of observations and for each state it calculates the most probable preceding state. Thus, we need to calculate all the state probabilities for one observation before we can move to the other, which necessitates a barrier. We provide details of parallel implementations of this algorithm in Section 4.1.2.

MapReduce - This dwarf characterizes the repeated independent execution of a function and the aggregation of the results at the end of the execution. Nearly no communication is required between processes. A map function processes a key/value pair to generate a set of intermediate key/value pairs, and then a reduce function merges all intermediate values associated with the same intermediate key. To demonstrate this dwarf, we use an algorithm that processes a text file and maps all distinct words with the number of their occurrences in the text.

Work-sharing cannot be directly applied for this algorithm because partition boundaries have to be properly adjusted to accommodate partitions whose boundaries break the words. Thus, shared-memory implementations are based on tasks. In addition, although there is a straightforward dependency between the *Map* and *Reduce* functions, our experiments have showed that in a shared memory system, given an arbitrary partition, the *Reduce* function needs to be executed immediately after the *Map* function to preserve cache locality. Therefore, our dataflow implementations are equivalent to the fork-join parallelism of shared memory implementations.

NBody methods – The algorithms from this dwarf involve interactions between many discrete points in space. The Barnes-Hut and Fast multipole algorithms hierarchically combine forces or potentials from multiple points to reduce the computational complexity of the direct particle-particle methods. These algorithms apply divide-and-conquer strategy to recursively divide the space into multiple subspaces, thus forming quadtrees or octrees, and treating a distant group of points as a single point, thus reducing the computational complexity of force calculations.

The DaSH benchmark for this dwarf implements the Barnes-Hut algorithm [5]. The parallelization is done by spawning tasks on a desired level of the Barnes-Hut tree that divides the 3-D space until all cubes contain at most one body. Given the Plummer distribution of bodies that we use for our tests, the tree is highly irregular, which means that the subtrees on a given level have different loads. Therefore, spawning tasks on deeper levels of the tree does not help the performance because many tasks end up processing cubes with just a few bodies, while others process cubes with much larger number of bodies. This introduces an overhead of handling a large number of short tasks, while at the same time providing little in terms of properly partitioning and balancing the work. The Barnes-Hut algorithm essentially depends on barriers that separate tree construction, force calculation and advancing of the bodies in each iteration. Thus, dataflow implementations apply the fork-join approach similar to the task-based shared memory implementations.

Sparse linear algebra – This dwarf represents algorithms that are used when input data sets (vectors and matrices) have a large number of zero entries. The dependence structure of these algorithms tends to be very complex in order to avoid operations on zero entries. This dwarf is represented in DaSH with the benchmark that implements a sparse LU decomposition of a matrix that is stored in a block compressed storage format. Shared memory implementations are based on work-sharing and implicit barriers that separate steps of execution in order to model the complex dependency structure of algorithm operations. Conversely, these dependencies are naturally expressed using dataflow, which eliminates the need for barriers and allows a straightforward implementation of the LU algorithm.

Spectral methods - This dwarf features a class of techniques from applied mathematics and scientific computing to numerically solve certain differential equations. As an example, we use the 3D FFT algorithm where basic 1D FFT transformations are handled by the *sfftw* library. The steps of the algorithm are inherently separated with barriers, which means that work-sharing implementations are better suited for this dwarf than dataflow.

Structured grid – The algorithms from this dwarf perform computations on regular multidimensional grids. Computation proceeds as a sequence of grid update steps. In each step, a given function is applied in parallel to all nodes in the grid, typically using the data from neighbouring nodes. DaSH benchmark for this dwarf implements Gauss-Seidel stencil computation on a 2-D square grid.

Shared memory implementations apply block-based wavefront parallelization pattern, while dataflow implementations organize block calculations in a dependency graph that resembles a wavefront pattern. However, in shared memory implementations there is an implicit barrier between each step of the wavefront execution as part of the parallel for loop that schedules the execution of blocks from a given step to worker threads. Moreover, there is also a barrier between successive iterations of the algorithm. Dataflow implementations avoid these barriers and only synchronize at every *n-th* iteration to test for the convergence. Given that solutions typically require a few thousand iterations or more to converge to a satisfactory level, we can safely perform convergence test after ten or twenty iterations, instead of doing the test in each iteration. We apply this approximation in all implementations.

Unstructured grid – This dwarf contains algorithms that perform updates on unstructured meshes that usually model irregularly shaped objects. Computation proceeds as a sequence of mesh update steps. Updates typically involve multiple levels of memory references, because the update of each point requires to first determine a list of neighbouring points and then to load values from them. The main problem in parallel algorithms for this dwarf is efficient partitioning of data that can provide both balanced workloads and minimal communication. The algorithm that represents this dwarf in DaSH is the Jacobi stencil computation on an unstructured triangular mesh grid. The input that we use represents a reconstruction of a photo of the Dragon from Stanford Computer Graphics Laboratory repository [22]. We use METIS library [15] to partition the input. In all implementations, we perform convergence test after *n* iterations instead of doing the same test after each iteration.

In shared memory implementations, we spawn a task for each partition and wait until all tasks have finished before we continue with the next iteration. Thus, there is a barrier between successive iterations of the algorithm. Dataflow implementations avoid the barriers by organizing calculations of different partitions based on their mutual dependencies. We deduce these dependencies using the adjacency list and partition lists provided by METIS to find all edges that connect different partitions. If there is an edge connecting two partitions then there is a dependency between them. Thus, we build a dependency graph between partition tasks, each of which can have different number of dependencies.

3.1. The DaSH Benchmark Suite Properties

Our decision to base the design of the DaSH benchmark suite on Berkeley dwarfs provides comprehensive variety of algorithms and application domains. In addition, most of the benchmarks are not trivially parallelizable, because they are either irregular or their characteristics change during the execution. Thus, DaSH is characterized with *breadth*.

For each benchmark, DaSH includes sequential, two shared memory implementations (OpenMP and TBB) and three hybrid dataflow implementations (OmpSs, ADF and TBB Flow Graph framework). Shared memory implementations are further based on work-sharing, tasking or a combination of these two. Thus, DaSH is also characterized with *depth*.

Moreover, different execution aspects of DaSH benchmarks can be controlled using command line parameters, such as the size and the nature of the problem (e.g. the particle distribution in *NBody methods*). Also, the parameters of the parallelization can be changed, such as the number of tasks or partitions that an application will generate and the number of threads in execution. DaSH already comes with two sets of input files, *small* and *large*, but the user can generate additional input files using a supplied input generator application. Ultimately, DaSH is written in C/C++, which also makes it *portable*.

4. PROGRAMMABILITY EVALUATION

To demonstrate the practical value of the DaSH benchmark suite for the research of hybrid dataflow models, we first asses programmability of these models compared to traditional shared memory models. We discuss two programmability aspects: first, we analyse cases in which dataflow provides better support for algorithm implementation and then we compare code complexities of different parallel implementations.

4.1. Algorithm Support

The value of dataflow is that it provides a natural support for implementation of applications characterized with complex or irregular dependency structure between their operations. Traditional shared memory implementations of such applications are feasible; however, they typically rely on artificial barriers to ensure a correct order of operations, which limits parallelism. On the contrary, dataflow allows a straightforward implementation of these applications because dependencies between operations can be easily expressed in this model. The runtime system ensures that operations are executed if and only if their dependencies are satisfied. DaSH benchmarks that have these application characteristics are *Dense algebra*, *Sparse algebra*, *Structured grid* and *Unstructured grid*. In this section, we compare parallel implementations of the *Dense algebra* benchmark to illustrate the value of dataflow.

Furthermore, we show that dataflow can be useful even if barriers are essential to the algorithm. Two DaSH benchmarks have such characteristics: *Dynamic programming* and *Graphical models*. We use the latter to illustrate the idea.

4.1.1. Dense Algebra

The DaSH benchmark that represents this dwarf is an implementation of the block Cholesky decomposition – Figure 2. The algorithm decomposes positive-definite matrix A into the product of the lower triangular matrix and its conjugate transpose $A = LL^T$. Block operations are performed by calling corresponding CLAPACK functions: *spotrf* that computes the Cholesky factorization of a block, and *ssyrk*, *strsm* and *sgemm* that solve a matrix equation or perform one of the rank k operations on blocks.

```
float ***A;
void sgemmWrap(...) { sgemm(...);}
void ssyrkWrap(...) { ssyrk(...);}
void strsmWrap (...) { strsm (...);}
void spotrfWrap (...) { spotrf (...);}

void SolveOpenMP()
{
  for (long j = 0; j < NumBlocks; j++) {
    /* Cholesky Factorization of block A[j,j] */
    spotrfWrap(A[j][j], BlockSize);

    #pragma omp parallel for firstprivate(j)
    for (long i = j+1; i < NumBlocks; i++) {
      /* A[i,j] <- A[i,j] = X * (A[j,j])^t */
      strsmWrap( A[j][j], A[i][j], BlockSize);
    } /* end parallel for */

    #pragma omp parallel
    {
      #pragma omp single
      {
        for (long i = j+1; i < NumBlocks; i++) {
          #pragma omp task firstprivate(i, j)
          {
            for (long k= 0; k< j; k++) {
              /* A[i,j] = A[i,j] - A[i,k] * (A[j,k])^t */
              sgemmWrap( A[i][k], A[j][k], A[i][j], BlockSize);
            }
          } /* end task */
        }

        #pragma omp task firstprivate(j)
        {
          for (long i = 0; i < j; i++) {
            /* A[j,j] = A[j,j] - A[j,i] * (A[j,i])^t */
            ssyrkWrap(A[j][i], A[j][j], BlockSize);
          }
        } /* end task */

      } /* end single */
    } /* end parallel */
  } /* end for */
}
```

a)

```
void SolveADF()
{
  for (long j=0; j<NumBlocks; j++) {
    #pragma adf task trigger_set(ssyrkToken[j]) execute(1)
    {
      spotrfWrap(A[j][j], BlockSize);
      spotrfToken[j] = j;
    } /* out spotrfToken[j] */

    #pragma adf task trigger_set(strsmToken[j]) relaxed execute(j-1)
    {
      ssyrkWrap( A[j][strsmToken[j]], A[j][j], BlockSize);
      if (strsmToken[j] == j-1)
        ssyrkToken[j] = j;    /* generate output only once */
    } /* out ssyrkToken[j] */

    for (long i = j+1; i < NumBlocks; i++) {
      #pragma adf task trigger_set(spotrfToken[j], sgemmToken[i][j])
                       relaxed execute(1)
      {
        strsmWrap( A[j][j], A[i][j], BlockSize);
        strsmToken[i] = j;
      } /* out strsmToken[i] */

      #pragma adf task trigger_set(strsmToken[j], strsmToken[i])
                       relaxed execute(j-1)
      {
        long k = strsmToken[j];
        sgemmWrap( A[i][k], A[j][k], A[i][j], BlockSize);
        if (k == j -1)
          sgemmToken[i][j] = j;
      } /* out sgemmToken[i][j] */
    }

    #pragma adf task relaxed execute(1)
    sgemmToken[j][0] = 0; /* initial sgemmToken[j][0] for column 0*/
  } /* end for */

  #pragma adf task relaxed execute(1)
  ssyrkToken[0] = 0; /* initial ssyrkToken[0] for block A[0][0]  */

  #pragma adf_start
  #pragma adf_taskwait
}
```

b)

Figure 2: Block Cholesky decomposition: a) OpenMP implementation, b) ADF implementation.

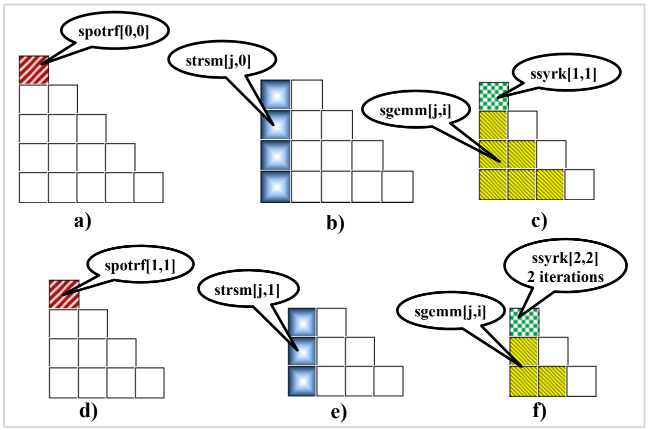


Figure 3: An example of the OpenMP Cholesky decomposition.

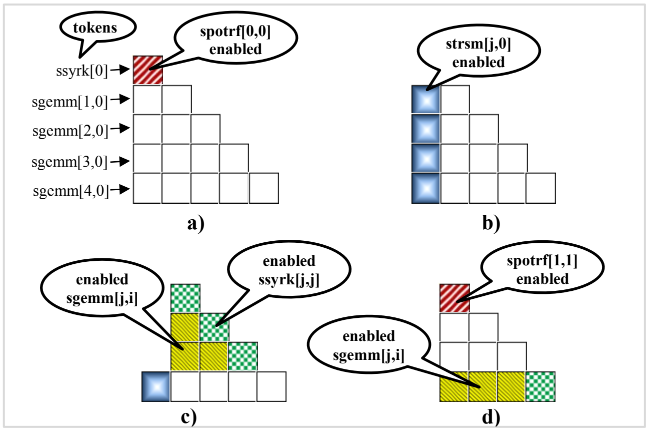


Figure 4: An example execution of the ADF Cholesky decomposition

In the OpenMP solution, at the start of each iteration, a single thread performs *spotrf* operation on a diagonal block – Figure 3a and 3d – while other threads are idle. Next, *strsm* operations in the corresponding column are executed in parallel – Figure 3b and 3e. Then, a single thread spawns a single *ssyrk* task and a number of *sgemm* tasks for the blocks bellow the diagonal – Figure 3c and 3f. These tasks can be performed in parallel; however, a single thread has to perform *ssyrk* operation. The alternative is to use synchronization, but this would result in extreme overhead if transactional memory is used for that purpose.

The dataflow approach provides a more elegant solution which follows the nature of the algorithm. First, we do not need to maintain barriers. Rather, we can orchestrate the execution based on data dependencies between block operations. Figure 2b shows the ADF implementation and Figure 4 illustrates the idea. Initially we produce *ssyrk[0]* token that enables *spotrf[0,0]* task, and *sgemm[j,0]* tokens for all the blocks from column 0 – Figure 4a. When *spotrf[0,0]* task finishes its execution, it produces *spotrf[0]* token that enables all *strsm* tasks from column 0 – Figure 4b. As these tasks execute, they produce *strsm[j]* tokens. Token *strsm[j]* enables *ssyrk[j,j]* task, while all *strsm[j]* tokens enable all *sgemm* tasks from the part of the matrix bellow diagonal, excluding column 0 – Figure 4c. Task *ssyrk[1,1]* then produces token *ssyrk[1]*, while *sgemm[1,i]* tasks produce corresponding tokens, which initiates the same sequence of operations for the next iteration – Figure 4d.

Evidently, this approach allows more tasks to overlap and thus results in better concurrency. For example, in the OpenMP

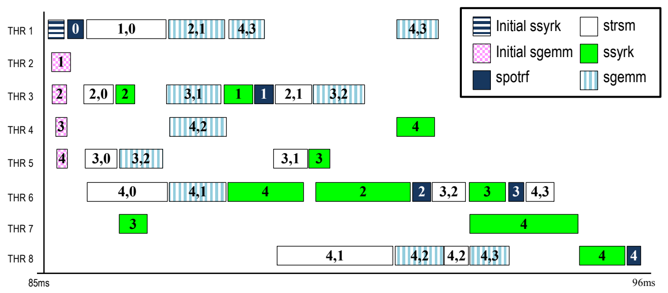


Figure 5: Execution trace of the ADF Cholesky decomposition of the 5x5 block matrix.

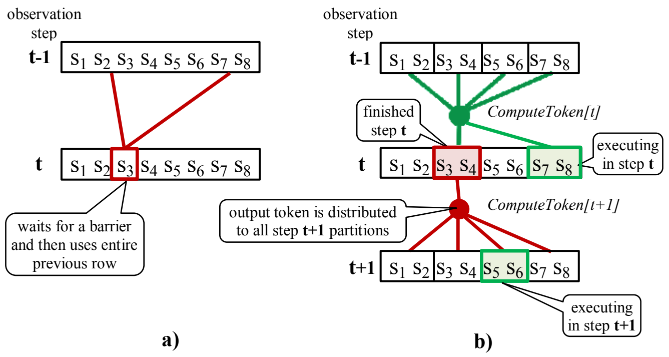


Figure 6: Dependencies and execution of graphical modelsbenchmark: a) shared memory implementation, b) dataflow implementation.

solution, *ssyrk[4,4]* block has to wait for all previous iterations and only then it is able to execute its operation using all blocks from rows 3 and 4 and previous four columns. Instead, in the ADF implementation, as soon as *strsm[j,3]* and *strsm[j,4]* tasks are finished, *ssyrk[4,4]* can execute its operation using these two blocks. Similar reasoning applies to *sgemm* tasks. Figure 5 shows the trace of the program execution with 5x5 blocks, obtained using the Paraver performance analysis toolchain [4]. We have added the annotations to identify each of the tasks.

4.1.2. Graphical Models

In Section 3 we have explained that Viterbi algorithm iterates through a sequence of observations and for each state it calculates the most probable preceding state. Thus, we need to calculate all the state probabilities for one observation before we can move to the other, which essentially requires a barrier – Figure 6a. Still, using dataflow it is possible to avoid this barrier – Figure 6b. Specifically, we partition each row to *n* partitions, where *n* is the number of worker threads, and build dependency graph so that each partition from the observation step *t* is processed by a dedicated task *ComputeTask[t][j]*, where *j* corresponds to the position of the partition in the row. Each *ComputeTask[t][j]* has an input dependency on *ComputeToken[t-1]* that is produces by all *Compute* tasks from the previous observation step. Importantly, *ComputeTask[t][j]* should produce output data if and only if it has executed *n* times, once for each partition of states from the previous observation step. When *ComputeTask[t][j]* produces the output data, it enables the execution of all *ComputeTask[t+1][k],* for *k = 1..n* – Figure 6b.

However, this approach is not feasible when using OmpSs because the model builds the task dependency graph instead of the data dependency graph. For example, assume that task T_1 has an output dependency on data *X* and that task T_2 has an input dependency on *X*. Even if T_1 finishes the execution without even touching data *X,* the runtime system assumes that the task has produced a new value for *X* and thus enables the consumer task T_2.

	seq (Total)	omp	tbb	adf	tbb flow	ompss
Number of lines of code	3189	14%	13%	25%	37%	17%
Function count	242	1%	1%	9%	26%	4%
# lines of code per function	252	20%	20%	25%	24%	21%
Cyclomatic complexity	43.9	15%	12%	17%	21%	21%
Average number of tokens	1149.0	23%	26%	27%	35%	33%

Table 1: Parallel code complexity compared to sequential baseline.

Therefore, OmpSs does not allow selective output token generation which is necessary to implement the approach described here.

4.2. Code Complexity Analysis

Table 1 summarizes the data obtained comparing the code complexity of different implementations of the DaSH benchmarks using the *hfcca.py* tool [12]. Since DaSH benchmarks execute operations mostly by calling library functions, or they perform simple operations on large datasets, the number-of-lines-of code may be a misleading metric. Compared to the sequential implementation, on average we see 17-30% increase in number of lines of code of dataflow implementations, and only 14% increase in case of non-dataflow implementations. In absolute terms, this represents 50-70 additional lines of code on average. On the other hand, all parallel implementations increase the average number of tokens almost equally. This metrics is sometimes considered as a more reliable measure of the code size since it measures the number of control structures, variable names, and non-blank separators in the code.

In the extreme case, the TBB flow implementation of the *Unstructured grid* benchmark increases the code size by 268 additional lines. In this benchmark, although most of the partitions for our input mesh have up to 5 dependencies, there are also a few partitions with seven, eight and even nine dependencies, which complicates the implementation. Specifically, the ADF and OmpSs implementations have to provide a switch case for each number of possible input dependencies that a partition might have (the only output dependency is the partition itself). Even worse situation is in the TBB Flow implementation. Since the types of the function node and the corresponding join node that represents a task depend on the number of input dependencies that a partition has, we had to implement a factory construction pattern that builds these nodes. This means that we had to provide a class specialization for every possible number of input dependencies that a partition might have, which is not scalable. At least in the ADF model, this could be supported by an API extension that would enable supplying an arbitrary number of dependencies to a task using C++ vectors.

Dataflow implementations also increase the number of function calls slightly more than shared memory implementations; however, in terms of the remaining metrics, all parallel impleme-ntations have similar complexity. Since the DaSH benchmarks do not rely on synchronization heavily, the cyclomatic complexity metric that measures a number of linearly independent paths through a program's source code (lower the better) is also valid for our analysis. From Table 1 we see that also in this respect the parallel implementations have the same complexity. Overall, we can conclude that dataflow implementations are characterized with similar code complexity as shared memory implementations. Our experience in developing DaSH confirms this, since once we got accustomed to the dataflow paradigm, developing dataflow implementation usually required the same amount of time as developing their shared memory counterparts.

Dwarf	DaSH benchmark	irregular structure	shared memory		dataflow		programm-ability
			barrier required	TM synch.	barrier required	TM synch.	
branch & bound	Traveling Salesman	●		moderate		moderate	dataflow tasking
dense algebra	Cholesky Factorization	●	●				dataflow
dynamic programming	Knapsack problem						dataflow work-sharing
finite state machine	Pattern search			light		light	tasking
graphical models	Viterbi algorithm		●				dataflow work-sharing
map reduce	Text mining						dataflow tasking
nbody methods	Barnes-Hut	●	●	light	●	light	tasking
sparse algebra	LU Factorization	●	●				dataflow
spectral methods	3D FFT		●		●		work-sharing
structured grid	Gauss-Seidel computation		●	light		light	dataflow
unstructured grid	Jacobi iterative method	●	●	light		light	dataflow work-sharing

Table 2: Summary of the programmability of the DaSH benchmarks.

Dwarf	Parameters	
	small	large
branch bound	500 cities	1000 cities
dense algebra	3072 x 3072 matrix , 24 x 24 blocks 128 x 128 block size	7680 x 7680 matrix , 60 x 60 blocks 128 x 128 block size
dynamic programming	3072 weights, max weight 500, knapsack capacity 6144	24576 weights, max weight 500, knapsack capacity 24576
finite state machine	file size 400MB pattern lenght 13 characters	file size 6.4GB pattern lenght 11 characters
graphical models	768 states, 40 observations, observation sequence length 200	3072 states, 50 observations, observation sequence length 200
map reduce	file size 40 MB	file size 160 MB
nbody methods	10k bodies, Plummer distribution, 50 steps, timestep 0.1	20k bodies, Plummer distribution, 50 steps, timestep 0.1
sparse algebra	3072x3072 matrix ,24x24 blocks 128x128 block size, null entries 79%	7680x7680 matrix ,60x60 blocks 128x128 block size , null entries 81%
spectral methods	3D matrix dimension 384	3D matrix dimension 768
structured grid	3072x3072 matrix, 1000 iterations, error step 5, block size 128	12kx12k matrix, 1000 iterations, error step 5, block size 256
unstructured grid	22998 nodes, 47794 triangles, 10000 iterations, error step 50	144647 nodes, 293232 triangles, 10000 iterations, error step 50

Table 3: Execution parameters of the DaSH benchmarks.

4.3. Summary of the DaSH Programmability

Table 2 summarizes our findings regarding the programmability of the DaSH benchmarks in different models. It further shows that no single parallel programming paradigm is suitable for all DaSH benchmarks. Still, dataflow is preferable for algorithms in which operations are dependent on each other in a way that prevents easy expression of the algorithm using work-sharing or tasking. Moreover, since modern hybrid dataflow models are based on tasks, they can readily be used for algorithms that require tasking model for efficient implementation.

5. PERFORMANCE ANALYSIS

For this evaluation, we have implemented the DaSH benchmarks using OmpSs, the ADF model and the TBB flow graph framework. As a reference, we use shared memory implementations of benchmarks in OpenMP and TBB programming models, which are based on work-sharing and/or tasks. We then compare the performance of these implementations with the sequential baseline. All applications are compiled using the gcc compiler version 4.7.2 that supports transactional memory (we have used default GCC-TM runtime configuration), except OmpSs implementations that are compiled using Mercurium compiler version 1.99. We also had to adapt the gcc implementation of the STL containers that we use in DaSH (vector, list and map) to avoid a negative performance impact of serialized transaction execution.

Table 3 contains execution parameters for each dwarf application. The experiments were conducted on Apple Mac Pro workstation with two 6-Core 64-bit Intel Xeon CPU X5650 Westmere processors running at 2.67GHz. Each processor unit has 12MB L3 cache memory. In addition, the cores are two way SMT-capable, giving a total number of 24 hardware threads. The machine is running Scientific Linux 6.2 (Carbon). For each experiment we report the average result of ten executions.

5.1. Experimental Results

Figure 7 and Figure 8 show maximum speedups achieved by parallel implementations compared to corresponding sequential baseline when large and small input data sets are used respectively. Evidently, the results support the view of the authors of Berkeley dwarfs [3] that it is unlikely that a single programming model can support efficient implementation of the whole application spectrum. Rather, each problem should be solved in the most appropriate way.

We first discuss the group of DaSH benchmarks that do not show a performance improvement when dataflow is used. In ***NBody methods*** and ***Spectral methods*** dwarfs, barriers are inherent to the algorithm and using dataflow does not increase the parallelism. In these cases, using traditional tasking or work-sharing seems to be the right choice. Similarly, in ***Branch and Bound*** dwarf, the key is to minimize the problem space by minimizing branching. Thus, the *Generator* task from our dataflow implementations works the best when only one or two new problems are generated for each task invocation. Using larger values for the parameter that controls the number of newly generated problems increases the problem space and decreases the performance. Thus, dataflow implementations work in a similar way as task-based shared memory implementations. Moreover, they perform the same, which means that both approaches provide equally good support for the implementation of this dwarf.

Common to ***Finite state machine*** and ***Map Reduce*** dwarfs is that, in all parallel implementations, the problem is partitioned to a number of independent tasks. Parallelism has a simple fork-join form. Although *MapReduce* is characterized with a straightforward dependency structure between *Map* and *Reduce* tasks, on our test machine the best results provides an implementation in which these functions are executed inside the same task. Threads that execute tasks reduce the data into their thread private dictionaries that are merged after the parallel section, in a sequential part of the execution. Figure 9 shows the results of various OpenMP and ADF implementations of the *MapReduce* dwarf that confirm these findings. *omp_thread* and *adf_thread* denote our default implementations that we have just described. In the *omp_global* and *adf_global* implementations reductions directly reduce into the global dictionary using synchronization, and in the remaining two implementations, *omp_part* and *adf_part*, each *Reduce* task creates a new partial dictionary. Furthermore, the *adf_part* implementation creates a number of instances of the *Map* and the *Reduce* tasks equal to the number of threads in the execution, and a single instance of the *Sum* task, which purpose is to collect all partial reductions generated by the *Reduce* tasks and merge them into a final

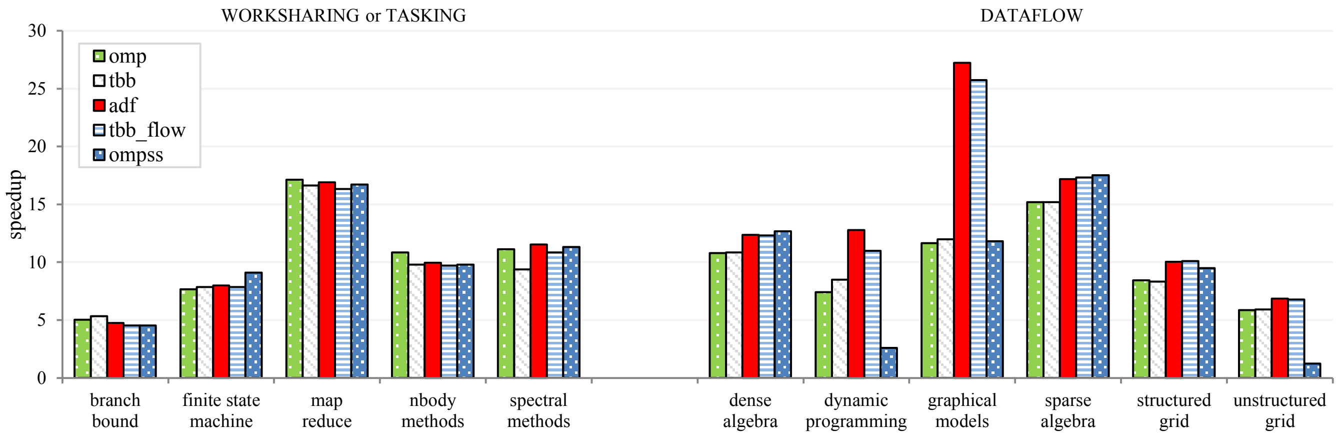


Figure 7: The maximum speedup of parallel DaSH benchmarks compared to the sequential baseline using the large input set. The maximum number of threads in execution is 24.

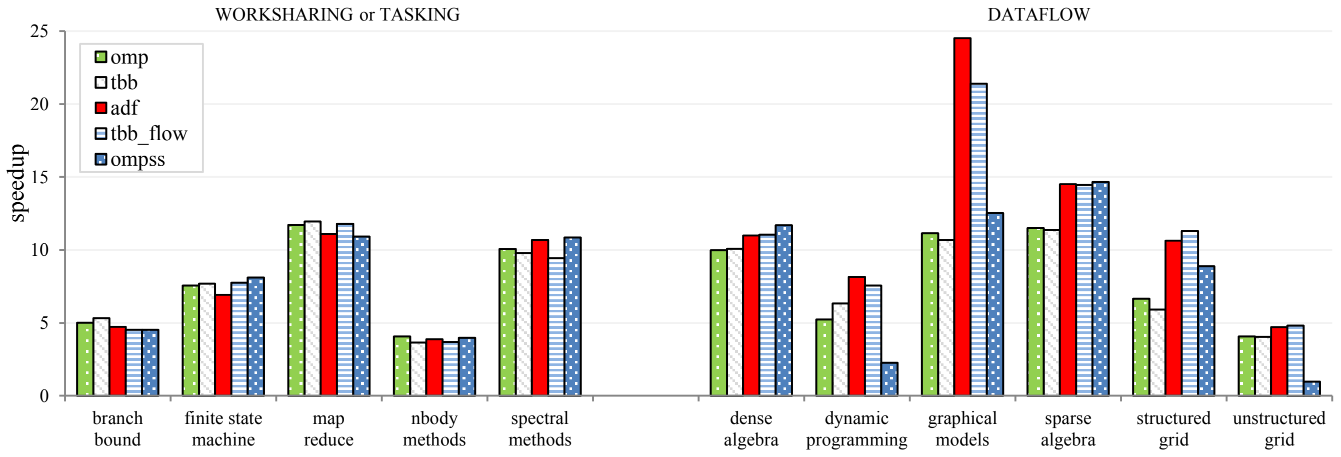
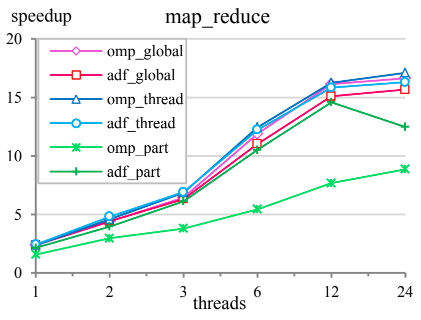
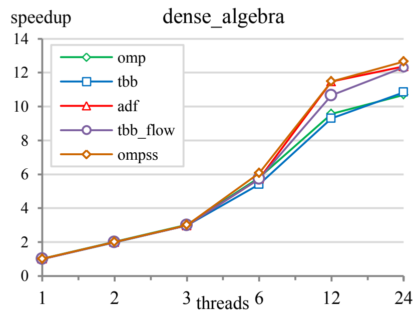


Figure 8: The maximum speedup of parallel DaSH benchmarks compared to sequential baseline using the small input set. The maximum number of threads in execution is 24.





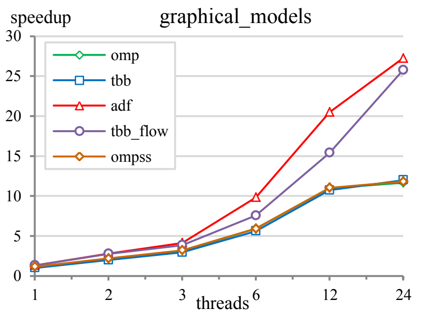


Figure 9: Scalability of the Map Reduce benchmark (large data set).

Figure 10: Scalability of the Dense Algebra benchmark (large data set).

Figure 11: Scalability of the Graphical Models benchmark (large data set).

dictionary. The *Sum* task performs merging in the parallel section rather than in the sequential part of the program. By executing only one instance of this task we eliminate the need for synchronization while building the final dictionary in parallel. We include all these implementations in DaSH to facilitate further testing on other test machines, which may provide different results.

The second group of DaSH benchmarks represents applications for which dataflow provides superior performance. DaSH benchmarks for ***Dense algebra***, ***Sparse algebra***, ***Structured grid*** and ***Unstructured grid*** dwarfs are examples that show the main value of dataflow, which is its ability to extract all the parallelism inherent in an application. While shared memory implementations rely on artificial barriers to implement a given algorithm from this group of dwarfs, which limits the parallelism and the performance, dataflow implementations support a straightforward implementation by directly mapping dependencies between algorithm operations into a task-based dataflow execution. Specifically, for these four dwarfs dataflow implementations provide on average up to 29.25% better speedup than the shared memory versions.

The remaining two benchmarks from the second group, ***Graphical models*** and ***Dynamic programming***, inherently depend on barriers to separate successive steps of the execution. Yet, in Section 4.1.2 we have showed that, using dataflow, it is possible to overlap the execution of tasks from different steps. Figures 7 and 8 show that this results in performance increase compared to shared memory implementations that depend on barriers. This is especially true for *Graphical models* dwarf for which dataflow implementations provide much better scalability - Figure 11 (note that the current OmpSs implementation is a pure shared memory implementation based on tasks, because our dataflow solution was not feasible using this model). Profiling shows that dataflow implementations have five times less L3 cache references, although all implementations result in almost identical number of L3 cache misses. Such a difference in the number of L3 cache references can be explained by the fact that in dataflow implementations, each task operates only on two row partitions, one from the current step and one from the previous step, while in the work-sharing implementations each task operates on a row partition from the current step and the entire row from the previous step of execution. This suggests that dataflow approach also improves the cache behaviour of a parallel implementation of this algorithm.

Comparing dataflow implementations, we notice that for both algebra dwarfs OmpSs implementations perform the best of all parallel implementations using both small and large datasets. For other benchmarks from the second group, implementations in the ADF model and the TBB Flow Graph framework alternately achieve the best performance. In two benchmarks, *Dynamic programming* and *Unstructured Grid*, OmpSs implementations have poor performance. This is caused by inefficiencies of the Mercurium compiler. In particular, we have measured the execution time of the sequential implementation compiled with Mercurium compiler and with gcc, which revealed that the former version runs almost five times slower than the latter. We suspect that dynamic memory allocation is the cause for this suboptimal performance because the problem gets worse when we increase the problem size. However, this should not be considered as a deficiency of the OmpSs model per se, but rather as the problem in the current implementation of the Mercurium compiler.

Figures 7 and 8 further show that parallel implementations of DaSH benchmarks perform consistently with different data sets. While a deeper analysis of the DaSH benchmark suite performance with extremely small or extremely large input data sizes is a part of our future work, current implementations show good stability given a significant difference in problem sizes of the small and large datasets used in this evaluation. Further, parallel implementations of the DaSH benchmarks achieve better speedup using the larger data set. In numbers, the aggregate average of maximum speedups of all parallel implementations using the large data set is 10.87, compared to 8.76x aggregate average when the small data set is used.

Finally, when we summarize the performance of all implementations in a specific model and compare it with the same value for OpenMP we find that, for the DaSH benchmarks, dataflow provides up to 23.87 % performance improvement for the large data set and 27.36% improvement for the small data set.

6. RELATED WORK

Due to its ability to express parallelism efficiently, dataflow model has been a focus of a number of recent research efforts. Beside the three hybrid dataflow models that we use in this paper, there are a number of alternatives. Microsoft TPL Dataflow Library [17] promotes actor-based programming by providing in-process message passing for coarse-grained dataflow and pipelining tasks in a similar way as the TBB flow graph framework.

Charm++ [14] supports directed acyclic graph execution and utilizes implicit message-passing to coordinate execution of actors across system nodes. Dooley et al. [7] have used Charm++ to study the memory-related issues of large-scale LU factorization. They propose a memory-aware scheduler in order to restrict the memory usage by limiting the concurrency. Data-Driven Multithreading (DDM) [23] is a model that provides dataflow scheduling of threads. In DDM a programmer is responsible for finding parts of the code that can execute in parallel and to explicitly express dependencies of each part. The programming interface is based on DFScala [10], which is a dataflow library for Scala. TIDeFlow [18] is a parallel execution model that is designed for efficient development of HPC programs for many-core architectures. The execution in TIDeFlow relies on shared memory to transfer data between actors that represent parallel loops and uses queues to distribute work among processors. SWARM [16] is another software runtime that aims to enable easier application scaling across numerous and diverse hardware components. The work in SWARM is organized in *codelets* that enable the runtime system to perform load-balancing decisions dynamically. The authors have implemented the Barnes-Hut algorithm using codelets and they report similar results using 24 cores, as we do in DaSH. Moreover, their implementation of the Graph500 benchmark consistently outperforms corresponding MPI implementation on a large scale. Software data-triggered threads (DDT) [24] is a programming model that relies on C pragma directives to declare data triggers for program threads. Using these pragmas, a programmer attaches a support thread to a variable that becomes a data trigger, upon which any changes to this variable spawn a new thread to execute associated support function. The evaluation of the SPEC2000 and Parsec benchmarks adapted to DDT shows promising results of this programming model.

To the best of our knowledge, DaSH is the first comprehensive benchmark suite for hybrid dataflow models. To prove the concept, existing hybrid dataflow models provide implementations of a few common benchmarks. Typically, these are matrix multiplication, Cholesky decomposition, LU and QR factorization, and other algebra applications. Amer et al. [1] have studied differences between data-driven and fork-join task parallelism execution of the Fast multipole method, suggesting that data-driven implementation can provide up to 22% better performance by avoiding the barriers and reducing the memory-bandwidth. Furthermore, Seaton et al [21] have extended the DFScala with TM support and show that such model can be efficiently used in parallelization of Lee's algorithm for circuit routing. Still, these applications are standalone and not part of a larger suite. Therefore, their coverage of algorithms and application patterns is limited. On the other hand, DaSH covers a wide range of algorithms and application domains and features not only the applications that are suitable for dataflow, but also those for which other forms of parallelism may be more appropriate.

7. CONCLUSION

In this paper we have presented DaSH – the first comprehensive benchmark suite for hybrid dataflow models. DaSH covers a wide range of application domains and provides two input data sets and the tool for generating new input data that should facilitate comprehensive evaluation of the benchmarks that comprise the suite. Many aspects of benchmarks' behaviour and their parallelization can be further controlled using command line parameters. Therefore, DaSH is characterized with three important properties: breadth, depth and portability.

Using DaSH for the evaluation of the OmpSs, ADF and TBB Flow Graph models, we have identified two main benefits that a hybrid dataflow model can offer. First, it can provide a straightforward implementation of certain types of irregular algorithms, which follows only logical dependencies present in the algorithm, and does not restrict the concurrency by inserting unnecessary barriers. And second, we have identified applications for which dataflow provides barrier-free implementation even when the algorithm inherently depends on barriers.

Overall, hybrid dataflow models are a promising alternative to traditional shared memory programming models. However, it is important for one such model to provide good support for work-sharing. In addition, we have shown that current API support, provided by the three hybrid dataflow models that we use in this paper, is not sufficient for an elegant implementation of an application characterized by tasks that can have an arbitrary number of dependencies. We have suggested that an API extension that would enable supplying an arbitrary number of dependencies to a task using C++ vectors could solve this problem. Finally, we have shown that dataflow models based on data-dependency graphs provide more flexibility compared to OmpSs that builds a task dependency graph, but also that OmpSs provides better programmability and performance for algebra problems.

We plan to continue this research by extending the list of models and applications included in DaSH. We have released the current version of the suite to support the research of hybrid dataflow models. Given the recent publication of the latest OpenMP 4.0 standard that introduces task dependency into its tasking model makes DaSH appealing for users who wish to adopt this new paradigm.

8. ACKNOWLEDGMENTS

We thankfully acknowledge the Microsoft Research support though the BSC-Microsoft Research Center, the European Commission through the HiPEAC-3 Network of Excellence, the Spanish Ministry of Education (TIN2007-60625, and CSD2007-00050), the Generalitat de Catalunya (2009-SGR-980) and the ERC Advanced Grant through the RoMoL project (Grant Agreement 321253).

REFERENCES

1 Amer, A., Maruyama, N., Pericàs, M., Taura, K., Yokota, R., and Matsuoka, S. Fork-Join and Data-Driven Execution Models on Multi-core Architectures: Case Study of the FMM. In *Proc. of the 2013 International Supercomputing Conference (ISC'13)* (Leipzig, Germany 2013), IEEE, 255-266.

2 Arvind and Culler, D. E. Dataflow architectures. In *Annual review of computer science*. Annual Reviews Inc., Palo Alto, CA, USA, 1986.

3 Asanovic, K., Bodik, R., Demmel, J. et al. A view of the parallel computing landscape. *Communications of the ACM*, 52, 10 (October 2009), 56-67.

4 Barcelona Supercomputing Center. *Paraver Performance Analysis Tool. January 2014. http://www.bsc.es/computer-sciences/performance-tools/paraver*.

5 Barnes, J. and Hut, P. A hierarchical O(NlogN) force calculation algorithm. *Nature*, 324, 4 (December 1986), 446–449.

6 Dennis, J. B. and Misunas, D. P. A preliminary architecture for a basic data-flow processor. *SIGARCH Computer Architecture News*, 3, 4 (1974), 126-132.

7 Dooley, I., Mei, C., Lifflander, J., and Kale, L. V. A study of memory-aware scheduling in message driven parallel programs. In *Proc. of the 17th International Conference on High Performance Computing (HiPC)* (Goa, 2010), IEEE, 1-10.

8 Gajinov, V., Stipic, S., Unsal, O. S., Harris, T., Ayguade, E., and Cristal, A. Integrating Dataflow Abstractions into the Shared Memory Model. In *Proc. of the 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)* (New York, 2012), IEEE, 243-251.

9 Gilmore, P. C. and Gomory, R. E. A linear programming approach to the cutting stock problem. *Operations Research*, 9, 6 (1961), 849-859.

10 Goodman, D., Khan, S., Seaton, C., Guskov, Y., Khan, B., Lujan, M., and Watson, I. DFScala: High Level Dataflow Support for Scala. In *the 2nd Workshop on Data-Flow Execution Models for Extreme Scala Computing* (Minneapolis, USA 2012), 18-26.

11 Harris, T., Larus, J., and Rajwar, R. *Transactional Memory (Second Edition)*. Morgan & Claypool Publishers, 2010.

12 *hfcca.py tool. January 2014. https://github.com/terryyin/hfcca*.

13 Intel. *Threading Building Blocks - version 4.2. October 2013. http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm#reference/reference.htm.*

14 Kale, L. V. and Krishnan, S. CHARM++: a portable concurrent object oriented system based on C++. In *Proc. of the 8th Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '93)* (New York, USA 1993), ACM, 91-108.

15 Karypis, G. and Kumar, V. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20, 1 (Decemebr 1998), 359-392.

16 Lauderdale, C., Glines, M., Zhao, J., Spiotta, A., and Khan, R. *SWARM: A unified framework for parallel-for, task dataflow, and distributed graph traversal*. ET International, Inc., Newark, USA, 2013.

17 Microsoft. *TPL Dataflow Library. January 2014. http://msdn.microsoft.com/en-us/library/hh228603.aspx*.

18 Orozco, D., Garcia, E., Pavel, R., Khan, R., and Gao, G. Tideflow: The time iterated dependency flow execution model. In *the 1st Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM)* (Galveston Island, USA 2011), IEEE, 1-9.

19 Perez, J. M., Badia, R. M., and Labarta, J. A dependency-aware task-based programming environment for multi-core architectures. In *Proc. of the 2008 International Conference on Cluster Computing* (Tsukuba. Japan 2008), IEEE, 142-151.

20 Rabiner, L. R. A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. *Proceedings of the IEEE*, 77, 2 (February 1989), 257 - 286.

21 Seaton, C., Goodman, D., Lujan, M., and Watson, I. Applying dataflow and transactions to Lee routing. In *the 5th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG)* (Paris, France 2012).

22 Stanford Computer Graphics Laboratory. *3D Scanning Repository. February 2014. http://graphics.stanford.edu/data/3Dscanrep/*.

23 Stavrou, K., Kyriacou, C., Evripidou, P., and Trancoso, P. Chip multiprocessor based on data-driven multithreading model. *International Journal of High Performance Systems Architecture*, 1, 1 (2007), 34-43.

24 Tseng, H. and Tullsen, D. M. Software data-triggered threads. In *Proc. of the International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'12)* (New York, USA 2012), ACM, 703-716.