Understanding Recursive Divide-and-Conquer Dynamic Programs in Fork-Join and Data-Flow Execution Models

Poornima Nookala
Illinois Institute of Technology
pnookala@hawk.iit.edu

Zafar Ahmad
Stony Brook University
zafahmad@cs.stonybrook.edu

Mohammad Mahdi Javanmard
Stony Brook University
mjavanmard@cs.stonybrook.edu

Martin Kong
University of Oklahoma
mkong@ou.edu

Rezaul Chowdhury
Stony Brook University
rezaul@cs.stonybrook.edu

Robert Harrison
Stony Brook University
robert.harrison@stonybrook.edu

***Abstract*—On shared-memory multicore machines, classic two-way recursive divide-and-conquer algorithms are implemented using common fork-join based parallel programming paradigms such as Intel Cilk+ or OpenMP. However, in such parallel paradigms, the use of joins for synchronization may lead to artificial dependencies among function calls which are not implied by the underlying DP recurrence. These artificial dependencies can increase the span asymptotically and thus reduce parallelism. From a practical perspective, they can lead to resource underutilization, i.e., threads becoming idle. To eliminate such artificial dependencies, task-based runtime systems and data-flow parallel paradigms, such as Concurrent Collections (CnC), PaRSEC, and Legion have been introduced. Such parallel paradigms and runtime systems overcome the limitations of fork-join parallelism by specifying data dependencies at a finer granularity and allowing tasks to execute *as soon as* dependencies are satisfied.**

In this paper, we investigate how the performance of data-flow implementations of recursive divide-and-conquer based DP algorithms compare with fork-join implementations. We have designed and implemented data-flow versions of DP algorithms in Intel CnC and compared the performance with fork-join based implementations in OpenMP. Considering different execution parameters (e.g., algorithmic properties such as recursive base size as well as machine configuration such as the number of physical cores, etc), our results confirm that a data-flow based implementation outperforms its fork-join based counterpart when due to artificial dependencies, the fork-join implementation fails to generate enough subtasks to keep all processors busy and does not have enough data locality to compensate for the lost performance. This phenomena happens when the input size of the DP algorithm is small or we have a huge number of compute cores in the system. As a result, with a fixed computation resource, moving from small input to larger input, fork-join implementation of DP algorithms outperforms the corresponding data-flow implementation. However, for a fixed size problem, moving the computation to a compute node with a larger number of cores, data-flow implementation outperforms the corresponding fork-join implementation.

***Keywords*-Dynamic Programming; Recursive Divide-and-Conquer; Data-Flow Parallel Model; Fork-Join Parallel Model; Concurrent Collections; OpenMP**

This work is supported in part by NSF grants CNS-1730689, CNS-1553510, CCF-1725428, and OAC-1931387.

I. Introduction & Motivation

A. Introduction

Dynamic Programming (DP) is an algorithm design technique that recursively decomposes a problem into smaller overlapping subproblems. It solves each unique overlapping subproblem exactly once and stores its result into the memory (DP table) for further reuse. Theoretically and practically, DP improves the performance of a recursive solution by preventing solving the repeating subproblems when they are encountered later [1, 2, 3]. DP algorithms can be viewed as trading off space-efficiency for reduced computation time [2]. DP is considered as one of the building blocks in solving a variety of combinatorial optimization problems [4]. It has numerous applications in different research and engineering areas, including computational biology [5], molecular modeling [6], etc.

The most common approach to implement DP algorithms is to use a loop-based program that populates the results into the underlying DP table cells iteratively. The recurrence relation of the DP specification enforces the correct ordering of storage and retrieval of the results of the subproblems. Such implementations often have good spatial locality and prefetching optimizations can be applied to gain further performance. However, they do not perform efficiently due to the lack of temporal locality. As a result, to overcome the shortcomings of the loop-based DP algorithms, researchers proposed tiled/blocked algorithms [7, 8, 9, 10] as well as standard 2-way recursive divide-&-conquer algorithms [11, 12]. Recursive divide-&-conquer DP algorithms are, unlike the tiled programs, cache oblivious [13, 12] and cache adaptive [14, 11]. Because of the heterogeneous nature of many modern supercomputers, standard 2-way (or any fixed r-way) recursive divide-&-conquer algorithms may suffer from the lack of performance portability and performance scalability on such

supercomputers. Such important limitations led to the introduction and development of parametric r-way recursive divide-&-conquer DP algorithms (r-way $\mathcal{R}$-$\mathcal{DP}$) to run efficiency on different architectures such as GPUs and distributed-memory parallel machines [15, 16, 17, 18, 19]

B. Motivation

On shared-memory multicore machines, classic 2-way algorithms have been implemented by fork-join based parallel programming paradigms such as Intel Cilk+ or OpenMP. However, in such parallel paradigms the use of joins for synchronization may create artificial dependencies among function calls which are not implied by the underlying DP recurrence. These artificial dependencies can increase the span asymptotically, and thus reduce parallelism [20, 21]. From a practical perspective, they can lead to resource underutilization, i.e., threads becoming idle. Due to such an important limitation, several researchers introduced task-based runtimes [22, 23, 24, 25, 26] and data-flow parallel paradigms [27, 28, 29, 30]. Such runtimes and paradigms which follow the data-flow model of execution and point-to-point synchronization, overcome the limitations of fork-join parallel paradigms. Data dependencies between tasks can be specified at finer granularity and tasks can execute as soon as the data becomes available (i.e., when dependencies are satisfied). In this paper, we investigate the application and efficiency of running DP algorithms on Intel Concurrent Collections (CnC) [31] which is one of the pioneering implementations of the data-flow based parallel paradigm. We compare the results with implementations in OpenMP. Considering different execution parameters (e.g., algorithmic properties such as recursive base size as well as machine configuration such as the number of physical cores, etc), we explain in what scenarios data-flow based implementation outperforms the fork-join based implementation. Considering Gaussian Elimination without pivoting (GE) algorithm, we provide an analytical model approximating the execution time of a DP computation. To summarize, followings are the **key contributions** of this work:

- By summarizing some of the important differences of fork-join based and data-flow based parallel paradigms, we explain how a standard 2-way recursive divide-&-conquer DP (2-way $\mathcal{R}$-$\mathcal{DP}$) algorithm is specified and developed in OpenMP and Intel CnC. We explain how the CnC runtime executes the program.
- We explain how the use of joins for synchronizations in the fork-join (OpenMP) implementations of $\mathcal{R}$-$\mathcal{DP}$ algorithms introduces artificial dependencies which leads to increase in span, reduction in parallelism and resource underutilization. We explain how data-flow implementation can resolve the issue.
- We design, implement and analyze three important DP benchmarks in OpenMP and Intel CnC: Gaussian Elimination without Pivoting, Smith-Waterman Local Alignment, and Floyd Warshall's All Pairs Shortest Path. We summarized the lessons learned from the experiments. We compared the experimental results and explained in what scenarios, each of the parallel paradigms outperform the other.
- Due to the importance of data movement cost in the memory hierarchy, in order to understand it better, for GE benchmark, we design an analytical model which correctly predicts the trend in data movement cost obtained from experimental results. The model can be easily extended to the other DP algorithms.

This paper is organized as follows. Sec. II provides a background on CnC model. Using the GE algorithm as a running example, Sec. III explains fork-join based implementation (in OpenMP) as well as data-flow based implementation (in Intel CnC) of the recursive divide-&-conquer DP algorithms. Experimental results are provided in Sec. IV. This section explains under what circumstances data-flow based implementation outperforms the fork-join based implementation and vice versa. Sec. V discusses the extensions and applications of the CnC model as related work. Sec. VI concludes the paper by summarizing the key points and mentioning the future work.

II. Background

Concurrent Collections (CnC) [28] is a data-flow based parallel programming model (originated from TStreams [32]). Different forms of parallelism, (including task, data, loop, pipeline and tree) can be expressed using this model. The important aspect of CnC is the idea of separation of concerns between application logic and parallel implementation. A CnC program/specification can be viewed as a communication means (or an interface) between the *domain expert*[1] and the *tuning expert*[2]. This separation of concerns simplifies the task of the domain expert, as writing a program in this language does not require any reasoning about parallelism or any knowledge of a target architecture [33]. The domain expert does not specify how operations are scheduled. The tuning expert (who can also be the domain expert) does not need to have an understanding of the domain (e.g., physics, chemistry, etc). S/he maps the CnC specification to a specific target architecture to be executed efficiently.

The three main CnC concepts are step collections, item collections (or data collections) and tag collections (or control collections). The CnC program is specified as a graph of collections, communicating with one another. More precisely, a CnC specification is a graph whose

[1]Whose interests and expertise in the application domain (e.g., finance, genomics, numerical analysis, etc) who does not necessarily have expertise in parallel programming and performance tuning.

[2]Whose interests and expertise are in performance and parallel programming.

nodes are either step collections, item collections, or tag collections, and the edges among them represent *producer*, *consumer*, and *prescription* dependencies. These edges enforce the partial order among the operations [33]. The relationships among the graph components are specified *statically* but during the execution, for each static collection, a set of *dynamic* instances are generated. A step collection corresponds to a specific computation (specified by the domain expert). A tag collection is the main concept for control flow of the program. Each tag collection is prescribed to a step collection, which means that putting tags into a tag collection will cause CnC runtime to generate an instance of the corresponding step collection, which will *eventually* execute with that tag instance as an input. Step collections dynamically read and write data through putting/getting items into/from item collections. From this perspective, the item collection can be considered as a placeholder for intermediate (or final) results produced and consumed by instances of the step collections. Listing 1 shows a simple example of a CnC specification [28].

```
1 /* tag collection myCtrl prescribes step collection
2   myStep */
3 <myCtrl>::(myStep);
4 /* step collection myStep consumes items from item
5   collection myData, produces item to myData and
6   puts tags into the tag collection myCtrl */
7 [myData] --> (myStep) --> [myData],<myCtrl>;
```

Listing 1: Simple CnC Specification

In the listing 1, paired parentheses represent step collections, paired square brackets represent item collections and finally tag collections are represented by paired angled brackets.

A graphical representation of the above program is shown in Fig. 1. In this representation, ovals represent step collections, rectangles represent item collections and Hexagons represent tag collections. The term *env* in the figure is the environment, which is the world *outside of the CnC program* and can be other threads or processes. They can put item(s) into (input) item collections and also trigger the computation by putting tag(s) into tag collections.

An instance of collections is identified by a unique tag. Item collections are an associative container that are indexed by the unique tags. The tags usually have meaning within the application, e.g., in a 2-D tiled computation, the tag $\langle i, j \rangle$ can represent the coordinates of a tile. CnC preserves the *dynamic single assignment*[3] property which is used in the proof of determinism of CnC programs[4] [28]. The C++ implementation performs dynamic (run-time) checks to ensure that the execution adheres to the single assignment rule. Budimlić et al. have argued that CnC programs are Turing Complete [28] though they do not claim

[3]Due to this property, the item with index (or tag) i, once written, cannot be overwritten.

[4]As long as step collections themselves are deterministic.

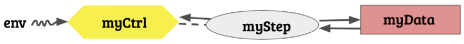

Figure 1: A graphical representation of the CnC program.

the absence of deadlocks. However, due to the deterministic property of CnC, deadlocks are straightforward to identify and fix.

CnC has different implementations in different languages including C++, Java, .NET, and Haskell. Since we use C++ implementations for our benchmarks, our focus will be on the C/C++ implementation of CnC which has two variations [33]. One is the X10-based [34] implementation from the Habanero project at Rice University. The other one is Intel's Concurrent Collections [31] which uses Intel's Threading Building Blocks (TBB).

The CnC implementation on TBB uses an object-oriented design methodology [33]. The runtime provides class definitions for the three collections (`TagCollection`, `StepCollection`, and `ItemCollection`). A CnC graph contains objects that represent the step collections, item collections, tag collections and their relationships. A user-defined C++ functor represents a step collection. When a tag is put in a tag collection, an instance of the prescribed step collection is created and mapped to a TBB task. The TBB task can be spawned immediately upon prescription or delayed until all the data dependencies are satisfied. Thus, the `Get` operation on an item collection can be blocking or non-blocking. In case of a blocking `Get`, if the item to be retrieved is not ready, the step collection instance is aborted and put in a separate list associated with the failed `Get` to be re-executed later. When an item becomes available, all the steps in the list, waiting for that item, get triggered to be executed. Data items are created and retrieved by calling the `Put` and `Get` methods of `ItemCollection`. The items are maintained in a TBB concurrent hash-map which are accessed by indices/tags.

III. Classic 2-way $\mathcal{R}$-$\mathcal{DP}$ Algorithms: Fork-Join based and Data-flow based Implementations

A. Overview

In this section, we explain two different implementations of the classic 2-way recursive divide-&-conquer DP algorithms (2-way $\mathcal{R}$-$\mathcal{DP}$) [12, 11]: fork-join implementation in OpenMP and data-flow implementation in Intel CnC. In this section, we explain how data-flow implementation eliminates the artificial dependencies which exist in the corresponding fork-join implementation. Elimination of such artificial dependencies leads to having finer-grained barriers in the execution of the algorithm. In Section IV, we discuss under what scenarios, having finer-grained barriers in the data-flow implementation can lead to a better parallelism and more efficiency over the corresponding fork-join implementation.

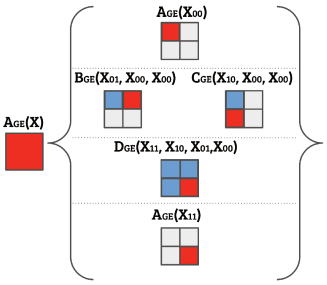

Figure 2: Function A of 2-way $\mathcal{R}$-$\mathcal{DP}$ GE algorithm.

For the rest of the paper, we use Gaussian Elimination without pivoting (GE) [3] as a running example which has DP-like structure. The GE algorithm has applications in Linear Algebra. It solves systems of linear equations and performs LU decomposition of symmetric positive-definite or diagonally dominant real matrices. A system of $(n-1)$ linear equations with $(n-1)$ unknowns $(x_1, x_2, ..., x_{n-1})$ is represented by a $(n \times n)$ matrix C. In such a matrix, the rth row represents the equation $\sum_{j=0}^{n-1}(C[r,j] \times x_j) = C[r,n]$. Listing 2 shows the loop-based serial implementation of the GE algorithm.

```
1 void I_GE(double **C,int N)
2  for(k=0; k < N-1; ++k)
3  for(i=0; i < N; ++i)
4  for(j=0; j < N; ++j)
5   if (i>k && j>=k) C[i][j]-=(C[i][k]*C[k][j])/C[k][k];
```
Listing 2: Loop-based serial implementation of GE

B. Fork-Join based Implementation of $\mathcal{R}$-$\mathcal{DP}$

Due to poor temporal locality of the loop-based implementation, which leads to poor I/O efficiency, researchers have introduced recursive divide-&-conquer algorithms (2-way $\mathcal{R}$-$\mathcal{DP}$). Such algorithms are theoretically and experimentally proven to be I/O efficient [11, 12]. Fig. 2 shows part of the classic 2-way $\mathcal{R}$-$\mathcal{DP}$ version of the GE algorithm. Computation starts with function A_{GE}. Function A_{GE} recursively calls itself for updating the top-left submatrix. Then it calls functions B_{GE} and C_{GE} in parallel to update the top-right and the bottom-left submatrices, respectively. Then function D_{GE} is called to partially update the bottom-right submatrix and finally function A_{GE} completes updates of the bottom-right submatrix. Functions B_{GE}, C_{GE}, and D_{GE} have similar recursive specifications. It has been shown that such an algorithm can be automatically generated from its corresponding loop-based code [11].

Listing 3 shows the OpenMP implementation of function A_{GE} depicted in Fig. 2. The fork-join based implementation provided in Listing 3 comes with a structural property that limits its performance: synchronization points among the recursive function calls, enforced by `#pragma omp taskwait` in OpenMP or `cilk_sync` in Intel Cilk+, in the join sections of the program create *artificial dependencies*

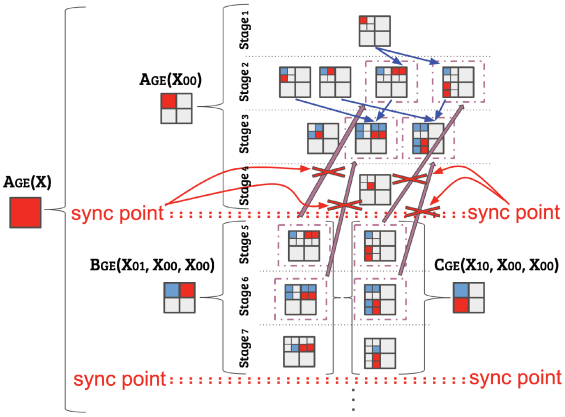

Figure 3: Barriers prevent further potential parallelism

among the function calls which are not implied by the underlying DP recurrence. These artificial dependencies also exist in the sub-function calls and hence increase the span asymptotically [20, 21] and thus reduce the parallelism. From a practical perspective, they can lead to *resource underutilization*, i.e., threads becoming idle. Fig. 3 illustrates the problem. In Fig. 3 synchronization points prevent function calls in stages 5 and 6 to be executed in stages 2 and 3. The same problem exists in recursive functions B_{GE}, C_{GE}, and D_{GE}.

```
1void func_A(double** X, int input_sz, int block_sz,
2            int base_sz, int i_lb, int j_lb, int k_lb)
3 if(block_sz <= base_sz) // base case part
4  for(int k = k_lb; k < k_lb+block_sz; ++k)
5  for(int i = i_lb; i < i_lb+block_sz; ++i)
6  for(int j = j_lb; j < j_lb+block_sz; ++j)
7   if(k < i && k < j)
8    X[i][j] -= (X[i][k] * X[k][j])/X[k][k];
9  return;
10 // recursive part
11 block_sz /= 2;
12 // Updating X11 (top-left sub-matrix)
13 func_A(X,input_sz,block_sz,base_sz,i_lb,j_lb,k_lb);
14 // In parallel, updating X12, and X21 sub-matrices
15 #pragma omp task
16 func_B(/*Updating X12, reading from X11*/);
17 #pragma omp task
18 func_C(/*Updating X21, reading from X11*/);
19 #pragma omp taskwait
20 func_D(/*Updating X22, reading from X11, X12, X21*/);
21 func_A(/*Updating X22*/);
```
Listing 3: OpenMP version of function A in $\mathcal{R}$-$\mathcal{DP}$ GE algorithm.

This problem has been identified and researchers have proposed *algorithmic solutions within the fork-join model* to resolve this problem statically [20, 21]. However, the proposed solutions are too complicated to develop, analyze, implement, and generalize. For example, they require hacking into a parallel runtime [21] or coming up with timing functions that are not straightforward [20]). On the other hand, task-based runtimes and data-flow parallel models provide a straightforward way to express the algorithms yet easily resolving the inefficiency introduced by synchronization points in the fork-join model. They overcome this limitation as follows. Data dependencies can be specified

directly at *finer granularity* and tasks get executed *as soon as its data dependencies are satisfied*. We explain Intel CnC implementation of the same algorithm in the next section.

C. Data-Flow based Implementation of $\mathcal{R}$-$\mathcal{DP}$

To implement the 2-way $\mathcal{R}$-$\mathcal{DP}$ in Intel CnC, by considering the recursive specification of function calls, we figure out the data dependencies among them. For example, from the specification of function A_{GE}, we can conclude that functions B_{GE} and C_{GE} depend on the output of A_{GE}. Similarly, function D_{GE} depends on the output of functions A_{GE}, B_{GE}, and C_{GE}.

The CnC program has four step collections with one for each of the functions, four tag collections with one for prescribing each of the step collections, and four item collections. The item collections are used as means of synchronization among the step collections to enforce *fine-grained data dependency* among the instances of step collections. The high level structure of the CnC graph/program is depicted in Listing 4.

```
1 struct GEContext: public CnC::context<GEContext> {
2  double* dp_table; int input_sz, base_sz;
3  typedef pair<pair<int,int>,pair<int,int>> CollectionT;
4  // defining step/tag/item collections, X in {A,B,C}
5  CnC::step_collection<FunctionX> funcX_step;
6  CnC::tag_collection<CollectionT> funcX_tags;
7  CnC::item_collection<CollectionT,bool> funcX_outputs;
8  // constructor, containging CnC graph information
9  GEContext(double* dp_t, int p_sz, int b_sz)
10  : dp_table(dp_t), input_sz(p_sz),
11    base_sz(b_sz),funcX_step(*this) {
12     // prescribing/producing/consuming relationships
13     // X in {A, B, C}
14     funcX_tags.prescribe(funcX_step,*this);
15     funcX_step.produces(funcX_outputs);
16     // consumes, defined based on the data dependencies
17     funcB_step.consumes(funcA_outputs);
18     funcB_step.consumes(funcD_outputs);
19     // ...
20  }
21 }
```

Listing 4: Intel CnC graph description of $\mathcal{R}$-$\mathcal{DP}$ GE algorithm.

In Listing 4, tag collections are templated by `CollectionT` which is `pair<pair<int,int>,pair<int,int>>`. This data structure contains the information which is needed for the functions to execute correctly. For example, for function B_{GE} which updates the tile $[I, J]$ of size b by reading from the tile $[I, K]$, the tag is `<<I,J>,<K,b>>`. Item collections are templated by `<CollectionT,bool>`, which is a mapping from the tile information `<<I,J>,<K,b>>` to Boolean indicating whether the tile has been updated completely (and it is ready to be used by other functions). For example, function B_{GE} puts the mapping (`<<`I_0`,`J_0`>,<`K_0`,`b_0`>>` $\rightarrow$ `true`) to the item collection `funcB_outputs` after completing the update on tile $[I_0, K_0]$. Such put will trigger the execution of all other functions waiting for this tile.

Step collections are templated by C/C++ structs `FunctionA`, ..., `FunctionD`. Each of these structs has a method called `execute` that takes the tag information `execInfo` as the first argument as well as the GE context `ctx` as the second argument.

Based on the data dependencies among the kernels we complete the implementation of the `execute` method in each of the structs. As an example, we explain method `FunctionD::execute` and others are implemented similarly. The implementation has been provided in Listing 5.

```
1 struct FunctionD {
2  /* Updating tile X by reading the tiles updated by
3     kernels C, B, and A */
4  int execute(const CollectionT& execInfo,
5              GEContext& ctx) const {
6   int I = execInfo.first.first,
7       J = execInfo.first.second,
8       K = execInfo.second.first,
9       block_sz = execInfo.second.second;
10   bool v;
11   if(block_sz <= ctxt.base_sz) { // base case
12    // checking write-write dependency
13    if(K > 0)
14     {ctx.funcD_outputs.get({{I,J},{K-1,block_sz}},v);}
15    // checking read-write dependencies
16    ctx.funcA_outputs.get({{K,K},{K,block_sz}},v);
17    ctx.funcB_outputs.get({{K,J},{K,block_sz}},v);
18    ctx.funcC_outputs.get({{I,K},{K,block_sz}},v);
19    // All dependencies OK, executing the base case
20    ge_iterative_kernel(ctx.input_sz, block_sz,
21                        I, J, K, ctxt.dp_table);
22    ctx.funcD_outputs.put({{I,J},{K,block_sz}}, true);
23   }
24   else { // recursive part
25    int tile_sz = block_sz/2;
26    for(int kk = 0; kk < 2; ++kk)
27    for(int ii = 0; ii < 2; ++ii)
28    for(int jj = 0; jj < 2; ++jj)
29     ctx.funcD_tags.put({{I*2+ii,J*2+jj},
30                         {K*2+kk,tile_sz}});
31   }
32   return CnC::CNC_Success;
33  }
34 };
```

Listing 5: Struct `functionD` in CnC implementation of $\mathcal{R}$-$\mathcal{DP}$ GE algorithm.

If the execution of function D_{GE} reaches its base case, it updates the tile/block with coordinate $[I, J]$ by *first* reading from the tiles/blocks with coordinate $[I, K]$, $[K, J]$, and $[K, K]$ which are produced by kernels C, B, and A, respectively. These three *read-write dependencies* can be enforced by using blocking `get` method on the item collections `funcC_outputs`, `funcB_outputs`, and `funcA_outputs`. Additionally, since it is updating the tile $[I, J]$, for $K > 0$, we need to ensure that the previous call to D_{GE} has finished its update on tile $[I, J]$. So, in order to enforce this *write-write* dependency, we use blocking `get` method on the item collection `funcD_outputs`. If all the dependencies are met, the kernel updates the tile/block and put the item `<<I,J>,<K,b>>`$\rightarrow$ `true` in the item collection `funcD_outputs`. Otherwise, if the function has not yet reached the base case, based on the recursive specification of D_{GE}, for *each of* the recursive function calls defined in its specification, irrespective of

their data dependencies[5], it puts tags into the tag collection `funcD_tags` to trigger their executions.

D. Improving Intel CnC performance through Tuners

Intel CnC provides tuners that can pass hints to the runtime system on how to improve performance [31]. One of them is the pre-scheduling tuner which enforces the execution of a step on the same thread that puts the prescribing tag, only after all the data dependencies are satisfied. This can improve performance by avoiding re-scheduling of a step due to unavailability of the items. Another way of improving the performance is to manually pre-declare all the dependencies, before the actual execution of updates in the algorithm. In this way, the underlying scheduler can trigger tasks when all the items are available. We have evaluated both approaches in order to tune the $\mathcal{R}$-$\mathcal{DP}$ computations and better understand the behavior.

IV. Experimental Results

We implement the three following benchmarks in Intel CnC and OpenMP: **(1) Gaussian Elimination without Pivoting (GE).** Section III-A contains a detailed explanation of this benchmark. It is noteworthy that the GE with partial pivoting does not have a DP-like structure [35] and going beyond DP algorithms is part of the future works. **(2) Smith-Waterman Local Alignment (SW).** The SW algorithm is used to determine the similarity between two DNA (or amino acid) sequences [36]. **(3) Floyd Warshall's All Pairs Shortest Path (FW-APSP).** For each pair of vertices in a directed graph, the FW-APSP algorithm computes the cost of the shortest path [3, 37].

A. Experimental Setup

The testbed for our experiments includes AMD Epyc and Intel Skylake processors which are part of the Mystic testbed [38]. The AMD Epyc 7501 machine has 2 sockets with 32 cores each, 8 NUMA zones, 32K L1, 512K L2 and 8192K L3 caches, 130GB RAM and per socket memory bandwidth of 170 GiB/s. The Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz machine has 8 sockets with 24 cores per socket, 8 NUMA zones, 32K L1, 1024K L2, 33792K L3, 768GB RAM and a theoretical memory bandwidth of 119 GiB/s.

For our Intel CnC implementations, we used Intel CnC version 1.0.1 and compiled using gcc version 7.5.0, with the following flags: `-std=c++11 -O3 -march=native -mavx2 -lcnc -lrt -ltbb -ltbbmalloc`. We have optimized the algorithms by eliminating branches in the innermost loop to enable vectorization. Naive implementation of SW uses ($\mathcal{O}(n^2)$) space and we have optimized the algorithm to consume ($\mathcal{O}(n)$) space for improving performance. GNU OpenMP implementations are used in the benchmarks with `OMP_PLACES =cores` and `OMP_PROC_BIND=close`. For Intel CnC experiments, we set `CNC_NUM_THREADS` to 64 on AMD Epyc and 192 on Intel Skylake servers.

[5]Note that all the data dependencies are enforced using the blocking `get` method

B. Performance Results

The goal of our evaluation is to characterize the behavior of $\mathcal{R}$-$\mathcal{DP}$ computations under a data-flow execution model. With this in mind, we designed $3 \times 2 \times 4 \times 4 = 96$ experiments, which include three benchmarks (GE, SW and FW-APSP) to explore on two multicore machines, while varying the problem parameters (problem size and base-case size). Our experiments show that even though $\mathcal{R}$-$\mathcal{DP}$ is meant to enhance the program's locality, controlling and characterizing the behavior in a data-flow model remains challenging. For each $\mathcal{R}$-$\mathcal{DP}$ benchmark, we implemented 4 versions:

- **(*Native-CnC*)** A base CnC program without scheduling hints.
- **(*Tuner-CnC*)** A CnC program with task scheduling hints by using CnC tuners (discussed in Sec. III-D).
- **(*Manual-CnC*)** A manually pre-scheduled CnC program (discussed in Sec. III-D).
- **(*OMP-Tasking*)** An $\mathcal{R}$-$\mathcal{DP}$ program using OpenMP tasking.

It is worth mentioning that we also implemented the benchmarks using non-blocking `get` approach [33] and noticed that the non-blocking `get` implementation is profitable only for smaller block sizes. However, the best overall performance is obtained by using blocking `get` approach.

Overall, our validation shows some high-level conclusions. First, $\mathcal{R}$-$\mathcal{DP}$ data-flow programs incur large runtime overheads on small block sizes. Second, large base case sizes reduce potential run-time task scheduling options.

Figures 4 and 5 show the execution time of the GE benchmark on the two machines. To understand the behavior of GE on these machines, due to the importance of the data movements in the memory hierarchy [39], we have developed an analytical model to estimate the overall cost of cache misses and the data movements.

As the first step, we will compute the total number of tasks generated by the recursive divide-and-conquer algorithm for GE. Observe that if the base case size is set to 1×1, the total number of times the base case is reached will be equal to the number of assignments made by the looping implementation of GE, which is: $\sum_{k=0}^{n-1}\sum_{i=k+1}^{n}\sum_{j=k+1}^{n}(1) = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n$. Now, if we coarsen the base case matrices to $m \times m$, clearly, the number of times such base cases will be reached, i.e., the number of base case tasks generated by the recursive algorithm, will be:

$$\frac{1}{3}\left(\frac{n}{m}\right)^3 + \frac{1}{2}\left(\frac{n}{m}\right)^2 + \frac{1}{6}\left(\frac{n}{m}\right) \tag{1}$$

Assuming a fair distribution of the tasks to the processors and among all the cores, we know that the total number of tasks per processor is $\left\lceil \frac{total\ number\ of\ tasks}{number\ of\ processors} \right\rceil$.

Once reached, a base case task that works on matrices of size $m \times m$ will perform between $\frac{1}{3}m^3 + \frac{1}{2}m^2 + \frac{1}{6}m$ (inside `func_A`) and $\sum_{k=0}^{m-1}\sum_{i=0}^{m}\sum_{j=0}^{m}(1) = (m+1)^2 m$ (inside `func_D`) assignments.

Considering the base-case implementation of the GE algorithm, which is the serial implementation (in the Listing 2), we can compute the maximum number of cache misses as follows. We know that the triply nested loop executes up to $\forall_{k=0}^{m-1}\forall_{i=0}^{m}\forall_{j=0}^{m}$ iterations, while accessing memory cells $C[i][j]$, $C[i][k]$, $C[k][j]$, and $C[k][k]$. Then, we proceed to count the total number of memory elements accessed for each distinct array reference, divided by the cache line size L, and add them up to get an upper bound on the total number of cache misses assuming that the cache cannot hold more than three cache lines and thus has very limited temporal locality. The bound is obtained as follows:

$$2\left(\sum_{k=0}^{m-1}\sum_{i=0}^{m}\left\lceil\frac{\sum_{j=0}^{m}8}{L}\right\rceil\right) + \left(\sum_{k=0}^{m-1}\sum_{i=0}^{m}1\right) + \left(\sum_{k=0}^{m-1}1\right) = m\left(1+(m+1)\left(1+\left\lceil\frac{8m+8}{L}\right\rceil\right)\right) \quad (2)$$

The first term in the summation above accounts for the maximum number of cache misses incurred when accessing $C[i][j]$ and $C[k][j]$, the second term accounts for $C[i][k]$, and the third one for $C[k][k]$. Given this, the total number of cache misses for each cache $L1$, $L2$, and $L3$, is approximated by adding up all the cache miss penalties at each level of cache. Figures 4 and 5 show the cost estimated using this model. The model assumes the recursion and looping overheads to be zero.

The ratio of the maximum cache misses estimated by the analytical model over the actual cache misses (i.e., $\frac{estimated\ max\ cache\ misses}{actual\ cache\ misses}$) provides an interesting measure of temporal locality. The larger this ratio the higher the temporal locality. For the GE benchmark with the problem size $8K \times 8K$, we captured the actual cache misses using the PAPI library [40] on SKYLAKE, and calculated this ratio. Table I shows the ratios for different base case sizes. Considering the sizes of $L2$ and per-core $L3$ cache share (which are 1MB and 32MB, respectively), we observe that for the $L2$ and $L3$ caches, this ratio sharply drops for the base cases larger than 128×128 and 1024×1024, respectively. These two base cases (128×128 for $L2$ and 1024×1024 for $L3$) reflect the largest blocks (more specifically, three such blocks storing double precision floats) that can fit into the $L2$ and $L3$ cache for GE on SKYLAKE.

Another important observation is that the execution times are significantly lower with hardware prefetching turned off for the CnC version. This is due to the coarse-grained data-flow irregularity not allowing full usage of prefetched data, i.e. the prefetcher bringing in data expected to be used, while (CnC) data-flow dependencies essentially flushing the cache immediately after, causing unnecessary overheads.

Base Size \ Cache Miss Ratio	L2 Cache	L3 Cache
64	107.61	294.50
128	240.63	660.02
256	38.38	1637.20
512	7.97	5793.74
1024	6.13	8247.60
2048	5.96	127.06

Table I: Ratio of the maximum estimated cache misses over the actual cache misses for the GE benchmark with problem size $8K \times 8K$ on SKYLAKE.

The analytical model does not take into account the load imbalance due to the data dependencies between the tasks causing the model to underestimate the cost. However, in some cases, using maximum cache misses to calculate the estimated cost causes the model to overestimate. The model also ignores overhead of scheduling of large number of tasks which significantly increases the execution time in case of *Manual-CnC*.

Another important observation from the figures is that for GE and FW-APSP benchmarks, for a fixed computation resource, as we increase the size of the input, the fork-join implementation (i.e,. OpenMP) outperforms the data-flow implementations (i.e., intel CnC). This is due to the fact that for the smaller problem size, because of the artificial dependencies that exists in the fork-join implementation, there are not enough tasks generated by the OpenMP to keep all the processors busy and does not have enough data locality. As a result, we have resource underutilization issue. However, as the problem size gets larger, in spite of the existence of the artificial dependencies, OpenMP is capable of generating enough tasks to feed all the processors and we have less resource underutilization.

Figures 6 and 7 show the execution time of SW benchmark on EPYC-64 and SKYLAKE-192 systems. Regarding the SW benchmark, the issue of artificial dependencies are so problematic that even for bigger problem sizes, still data-flow implementation outperforms. The main reason is the artificial dependencies in the fork-join implementation prevents the wavefront parallelism among the tasks, where tasks operate on tiles along diagonals of the input matrix[6].

However, data-flow implementation can easily benefit from the wavefront parallelism as data dependencies are specified at a finer granularity and there is no coarse-grain barrier synchronization for every wavefront computation.

[6]In fork-join implementation, there is a barrier synchronization for every wavefront computation

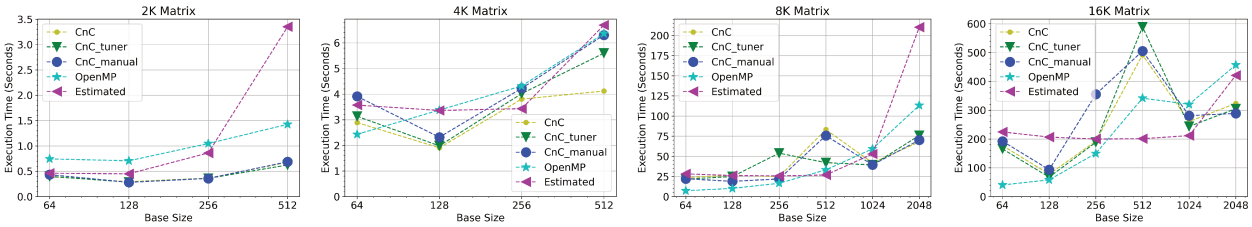

Figure 4: Execution time of Gaussian Elimination on EPYC-64

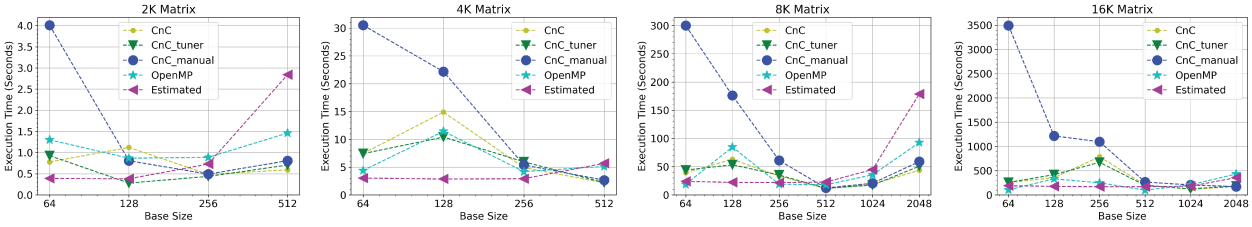

Figure 5: Execution time of Gaussian Elimination on SKYLAKE-192

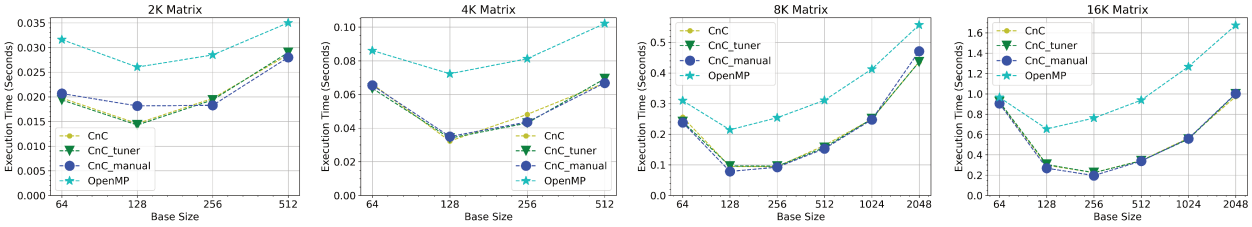

Figure 6: Execution time of Smith-Waterman on EPYC-64

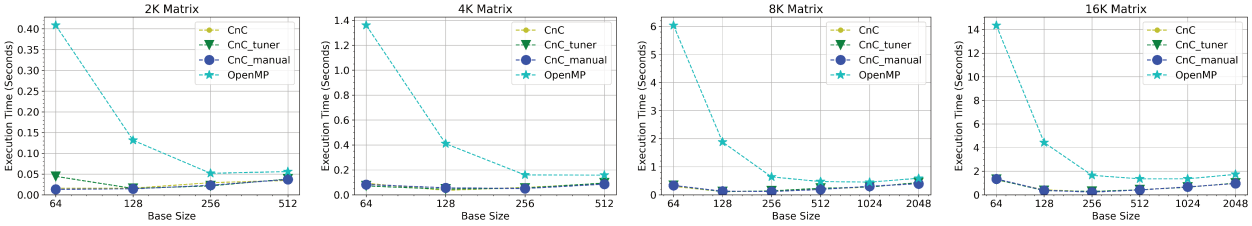

Figure 7: Execution time of Smith-Waterman on SKYLAKE-192

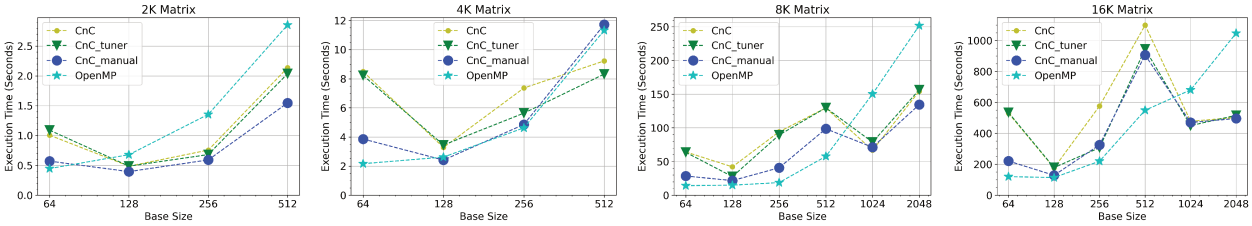

Figure 8: Execution time of Floyd Warshall's Algorithm on EPYC-64

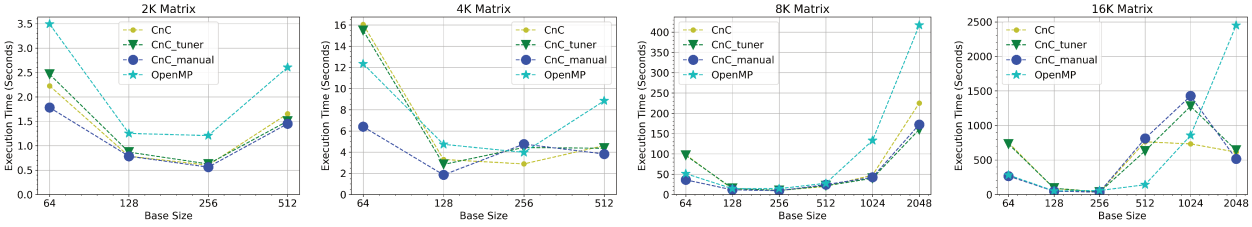

Figure 9: Execution time of Floyd Warshall's Algorithm on SKYLAKE-192

Figures 8 and 9 show the results obtained for FW-APSP benchmark. The analytical model described above for GE also applies to FW-APSP since both the same computational complexity as GE ($\mathcal{O}(n^3)$) and similar data access patterns.

Best running time is achieved with block size of 128 and 256 for all the variations of Intel CnC as well as OpenMP.

While fine-grained scheduling and task placement has not been explored in this work, we believe that leveraging other Intel CnC tuners such as `compute_on` and other forms of tasks pre-scheduling can lead to large performance improvements. Such tuner can effectively allow to pin specific tasks to execution locations (cores), thereby minimizing potential inter-core and inter-NUMA data movement.

V. Related Work

Sbirlea et al. introduced an intermediate graph representation for macro-data-flow programs (DFGR). It is an extension to the CnC model [41]. DFGR enables programmers to express programs at a high level with data-flow graphs as an intermediate representation. DFGR graphs consist of *step* nodes for computation and *item* nodes for data, which are partitioned into collections by unique tag. DFGR improves the efficiency by expressing what items are read and written to each step (through tag functions [42]). It is used as an abstraction to map the application for extreme-scale systems and run on heterogeneous architectures including GPUs/FPGAs, distributed-memory clusters, etc.

Later, they have proposed a polyhedral compiler framework, Data-Flow Graph Language (DFGL) which uses DFGR to represent dependencies [43]. The framework applies polyhedral analysis on dependencies to perform two important legality checks (single assignment rule and potential deadlocks) as well as applying automatic loop transformation, tiling, and code generation of parallel loops with coarse-grained and fine-grained synchronizations. DFGL framework compiles the input graph program into Habanero-C, which is an extension to C language built on top of CnC. The framework uses the ROSE compiler [44] to also generate OpenMP-4 compatible code, including task-level parallelism. They used Smith-Waterman, Cholesky factorization, Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) and some stencil kernels from PolyBench as their benchmarks. Their experimental results show that the DFGL versions optimized by their framework can deliver up to $6.9\times$ performance improvement relative to OpenMP versions of the benchmarks.

There are several experimental studies that have been done illustrating performance and scalability of the CnC model. Chandramowlishwaran et al. [45] evaluated two dense linear algebra algorithms: (1) asynchronous-parallel Cholesky factorization and (2) "higher-level" partly-asynchronous generalized eigensolver for dense symmetric matrices. For both benchmarks, they showed that their CnC implementations match or exceed the Intel Math Kernel Library (MKL) implementation. They also compared their CnC implementations with other parallel models including ScaLAPACK with shared-memory MPI, OpenMP, Cilk+, and PLASMA 2.0, on Intel Harpertown, Nehalem, and AMD Barcelona systems. For the C++ implementation, Budimlić et al. [28] has used Dedup, a benchmark from PARSEC benchmark suite [46] and compared the performance of CnC implementation and pthread implementation. They showed that the CnC implementation outperforms the pthread implementation for two reasons. First, in the pthread implementation, the load imbalance exists between the stages of the computation. Second, the pthread implementation, unlike the CnC implementation, has data locality (to a thread) issue. They also considered Cholesky Factorization as another case study and showed speed-up with respect to increase in the number of threads. Liu and Kulkarni implemented the proxy application, LULESH in CnC model and have shown that with step fusion and tiling optimizations, the implementation outperforms the original implementation with good scalability ($38\times$ speed up) for up to 48 processor machines [47].

VI. Conclusion

In this paper, using the GE algorithm as a running example, we discussed two different paradigms of parallel programming on shared-memory multicore machines: fork-join and data-flow parallel paradigms. Focusing on recursive DP algorithms, we explained the major performance bottleneck that exists in fork-join model: Joins at synchronization points introduce artificial dependencies which are not implied by the underlying DP recurrence.

These artificial dependencies exist in all sub-function calls and hence increase the span asymptotically and reduce parallelism. We explained how this performance issue can easily be eliminated by using data-flow based parallel paradigms such as CnC. Considering different execution parameters (e.g., algorithmic properties such as recursive base size as well as machine configuration such as the number of physical cores, etc), we explained under which scenarios a data-flow implementation outperforms the corresponding fork-join implementation. We provided an analytical model upper bounding the total data movement cost of the GE benchmark as one of the DP algorithms, which can be easily extended to the other DP algorithms. As the next steps, we would like to investigate algorithms beyond DP. Additionally, we would like to explore polyhedral compiler transformations which can help us automatically obtain data-flow based algorithms from the serial loop-based algorithms. Extending the framework to distributed-memory parallel machine is part of the future works.

Acknowledgments. The authors are grateful for the resources provided by the Mystic Testbed [38] for running the experiments. The authors would like to thank anonymous reviewers for their valuable comments.

References

[1] R. Bellman, "The theory of dynamic programming," Rand corp santa monica ca, Tech. Rep., 1954.

[2] S. S. Skiena, *The algorithm design manual: Text*. Springer Science & Business Media, 1998, vol. 1.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[4] V. T. Paschos, *Concepts of Combinatorial Optimization, Volume 1*. John Wiley & Sons, 2012.

[5] D. Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge university press, 1997.

[6] J. Klepeis, M. Ierapetritou, and C. Floudas, "Protein folding and peptide docking: A molecular modeling and global optimization approach," *Computers & chemical engineering*, vol. 22, pp. S3–S10, 1998.

[7] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A blocked all-pairs shortest-paths algorithm," *Journal of Experimental Algorithmics (JEA)*, vol. 8, pp. 2–2, 2003.

[8] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *PLDI'1991*, 1991, pp. 30–44.

[9] K. S. McKinley, S. Carr, and C.-W. Tseng, "Improving data locality with loop transformations," *ACM TOPLAS*, vol. 18, no. 4, pp. 424–453, 1996.

[10] F. Irigoin and R. Triolet, "Supernode partitioning," in *POPL'1988*, 1988, pp. 319–329.

[11] R. Chowdhury, P. Ganapathi, J. J. Tithi, C. Bachmeier, B. C. Kuszmaul, C. E. Leiserson, A. Solar-Lezama, and Y. Tang, "Autogen: Automatic discovery of cache-oblivious parallel recursive algorithms for solving dynamic programs," *ACM SIGPLAN Notices*, vol. 51, no. 8, pp. 1–12, 2016.

[12] R. A. Chowdhury and V. Ramachandran, "Cache-oblivious dynamic programming," in *SODA'2006*, 2006, pp. 591–600.

[13] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS'1999*. IEEE, 1999.

[14] M. A. Bender, R. Ebrahimi, J. T. Fineman, G. Ghasemiesfeh, R. Johnson, and S. McCauley, "Cache-adaptive algorithms," in *SODA'2014*, 2014.

[15] M. M. Javanmard, P. Ganapathi, R. Das, Z. Ahmad, S. Tschudi, and R. Chowdhury, "Toward efficient architecture-independent algorithms for dynamic programs," in *ISC-HPC'2019*. Springer, 2019, pp. 143–164.

[16] M. M. Javanmard, Z. Ahmad, M. Kong, L.-N. Pouchet, R. Chowdhury, and R. Harrison, "Deriving parametric multi-way recursive divide-and-conquer dynamic programming algorithms using polyhedral compilers," in *CGO'2020*, 2020, pp. 317–329.

[17] M. M. Javanmard, Z. Ahmad, J. Zola, L.-N. Pouchet, R. Chowdhury, and R. Harrison, "Efficient execution of dynamic programming algorithms on apache spark," in *CLUSTER'2020*. IEEE, 2020, pp. 337–348.

[18] M. M. Javanmard, "Parametric multi-way recursive divide-and-conquer algorithms for dynamic programs," Ph.D. dissertation, State University of New York at Stony Brook, 2020.

[19] M. M. Javanmard, P. Ganapathr, R. Das, Z. Ahmad, S. Tschudi, and R. Chowdhury, "Toward efficient architecture-independent algorithms for dynamic programs: poster," in *PPoPP'2019*, 2019, pp. 413–414.

[20] R. Chowdhury, P. Ganapathi, Y. Tang, and J. J. Tithi, "Provably efficient scheduling of cache-oblivious wavefront algorithms," in *SPAA'2017*, 2017, pp. 339–350.

[21] Y. Tang, R. You, H. Kan, J. J. Tithi, P. Ganapathi, and R. A. Chowdhury, "Cache-oblivious wavefront: improving parallelism of recursive dynamic programming algorithms without losing cache-efficiency," in *PPoPP'2015*, 2015, pp. 205–214.

[22] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *SC'2012*. IEEE, 2012, pp. 1–11.

[23] M. Kong, L.-N. Pouchet, P. Sadayappan, and V. Sarkar, "Pipes: a language and compiler for task-based programming on distributed-memory clusters," in *SC'2016*. IEEE, 2016, pp. 456–467.

[24] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical task-based programming with starss," *IJHPCA'2009*, vol. 23, no. 3, pp. 284–299, 2009.

[25] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar, "Habanero-java: the new adventures of old x10," in *PPPJ'2011*, 2011, pp. 51–61.

[26] G. Bosilca, D. Genet, R. J. Harrison, T. Herault, M. M. Javanmard, S. Brook, C. Peng, and E. Valeev, "Tensor contraction on distributed hybrid architectures using a task-based runtime system," 2018.

[27] A. Pop and A. Cohen, "Openstream: Expressiveness and data-flow compilation of openmp streaming programs," *TACO'2013*, vol. 9, no. 4, pp. 1–25, 2013.

[28] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[29] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[30] G. Bosilca, R. Harrison, T. Herault, M. Javanmard, P. Nookala, and E. Valeev, "The template task graph (ttg)-an emerging practical dataflow programming paradigm for scientific simulation at extreme scale," in *ESPM2'2020*. IEEE, 2020, pp. 1–7.

[31] "Intel (r) concurrent collections for c/c++," https://icnc.github.io/.

[32] K. Knobe and C. D. Offner, "Tstreams: A model of parallel computation (preliminary report)," Technical Report HPL-2004-78, HP Labs, Tech. Rep., 2004.

[33] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari, "Multi-core implementations of the concurrent collections programming model," in *CPC'2009*, 2009.

[34] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," *Acm Sigplan Notices*, vol. 40, no. 10, pp. 519–538, 2005.

[35] R. A. Chowdhury and V. Ramachandran, "The cache-oblivious gaussian elimination paradigm: theoretical framework, parallelization and experimental evaluation," *Theory of Computing Systems*, vol. 47, no. 4, pp. 878–919, 2010.

[36] T. F. Smith, M. S. Waterman *et al.*, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[37] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

[38] A. I. Orhean, A. Ballmer, T. Koehring, K. Hale, X.-H. Sun, O. Trigalo, N. Hardavellas, S. Kapoor, and I. Raicu, "Mystic: Programmable systems research testbed to explore a stack-wide adaptive system fabric," in *8th Greater Chicago Area Systems Research Workshop (GCASR)*, 2019.

[39] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Quantifying the energy cost of data movement in scientific applications," in *IISWC'2013*. IEEE, 2013, pp. 56–65.

[40] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with papi-c," in *Tools for High Performance Computing 2009*. Springer, 2010, pp. 157–173.

[41] A. Sbirlea, L.-N. Pouchet, and V. Sarkar, "Dfgr an intermediate graph representation for macro-dataflow programs," in *2014 Fourth Workshop on Data-Flow Execution Models for Extreme Scale Computing*. IEEE, 2014, pp. 38–45.

[42] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 61–70, 2012.

[43] A. Sbîrlea, J. Shirako, L.-N. Pouchet, and V. Sarkar, "Polyhedral optimizations for a data-flow graph language," in *LCPC'2015*. Springer, 2015, pp. 57–72.

[44] "The rose compiler project," http://rosecompiler.org/.

[45] A. Chandramowlishwaran, K. Knobe, and R. Vuduc, "Performance evaluation of concurrent collections on high-performance multicore computing systems," in *IPDPS'2010*. IEEE, 2010, pp. 1–12.

[46] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *PACT'2008*, 2008, pp. 72–81.

[47] C. Liu and M. Kulkarni, "Optimizing the lulesh stencil code using concurrent collections," in *WOLFHPC'2015*, 2015, pp. 1–10.