Electronic version of an article published as Pavel Zaichenkov et al, *Parallel Processing Letters,* Vol. 26 (3), 2016, 24 pages, DOI: http://www.worldscientific.com/doi/abs/10.1142/S0129626416500110

Parallel Processing Letters

The Cost and Benefits of Coordination Programming: Two Case Studies in Concurrent Collections and S-Net

Pavel Zaichenkov, Olga Tveretina, Alex Shafarenko

Compiler Technology and Computer Architecture Group
University of Hertfordshire, College Lane (STRI)
Hatfield, AL10 9AB, Herts, United Kingdom

Bert Gijsbers

Department of Applied Mathematics, Computer Science and Statistics
Ghent University, Krijgslaan 281 S9
9000 Ghent, Belgium

Clemens Grelck

Informatics Institute
University of Amsterdam, Science Park 904
1098 XH Amsterdam, Netherlands

Received February 2014
Revised January 2016
Communicated by Kemal Ebcioglu

ABSTRACT

This is an evaluation study of the expressiveness provided and the performance delivered by the coordination language S-Net in comparison to Intel's Concurrent Collections (CnC). An S-Net application is a network of black-box compute components connected through anonymous data streams, with the standard input and output streams linking the application to the environment. Our case study is based on two applications: a face detection algorithm implemented as a pipeline of feature classifiers and a numerical algorithm from the linear algebra domain, namely Cholesky decomposition. The selected applications are representative and have been selected by Intel researchers as evaluation testbeds for CnC in the past. We implement various versions of both algorithms in S-Net and compare them with equivalent CnC implementations, both with and without tuning, previously published by the CnC community. Our experiments on a large-scale server system demonstrate that S-Net delivers very similar scalability and absolute performance on the studied examples as tuned CnC codes do, even without specific tuning. At the same time, S-Net does achieve a much more complete separation of concerns between compute and coordination layers than CnC even intends to.

Keywords: performance measurement, coordination programming, stream processing

1. Introduction

Parallel programs are known to be hard to maintain and reuse. The reason for this is that they typically mix up two concerns of the kind that Dijkstra believed best to be kept separate: concurrency management in the broad and domain-specific computation in the narrow. This intertwining of code sections that implement the actual computing (or algorithm) with those that merely organize parallel execution (decomposition, communication, synchronization, etc.) is well known to significantly contribute to the challenge of parallel programming. Almost all commonly used parallel programming models, such as MPI, OpenMP, etc., share these characteristics.

Intel's Concurrent Collections (or CnC for short) aim at separating coordination from computation issues, but in our view does not achieve a complete separation of concerns in its concrete implementations. Consequently, coordination and computation code sections are still found next to each other in the source code.

We argue that S-Net [1, 2] achieves a much deeper separation of concerns than CnC by employing a dedicated coordination language (complete with its formal semantics and type theory) operating on top of domain-specific components. These components are written in a conventional language and remain entirely concurrency-agnostic. They not only dispense with all forms of concurrency internally, but they are likewise unaware of the provenance of data coming into a component and of the destination of data sent out of a component. Consequently, individual S-Net components are agnostic of their relative location within a larger S-Net streaming network, improving opportunities for code reuse. Components are even agnostic of the detailed composition of data other than that explicitly needed by a component to work.

An S-Net component is not allowed to hold an internal state and consequently cannot be used to synchronize messages (it would need to hold on to one while expecting another). That means that messages come to the component pre-synchronized by the coordination layer. Furthermore, a component can be transparently wound down and reinstated between messages. This makes S-Net especially flexible in a distributed computing context. On the other hand, the coordination program presents the coordination compiler with structures that can be analyzed to learn behavior modulo the behavior of the components. All of this offers attractive benefits for large-scale applications. There is one obstacle though in getting the coordination approach to work in practice: as it espouses a higher level of abstraction (which stems from its characteristic separation of concerns), practitioners tend to be wary of a possible loss of performance, especially in applications of the "no expense spared" type, where performance is the primary (if not the only) concern.

As the example of C++, where a higher level of abstraction *and* high performance happily co-exist, demonstrates, performance is not necessarily lost by the separation of concerns. This paper aims to demonstrate on two practical examples that aggressive structuring imposed by S-Net does not have to negatively affect the performance. In fact, we show that performance may, if anything, be better when

synchronization concerns (with their associated data management mechanisms) are removed from the components. That is the reason that we choose CnC as our basis for comparison: both systems use dataflow synchronization and single-assignment semantics for components. However, S-Net is a fully-fledged coordination language promoting automatic component configuration and reuse, whereas CnC requires a much deeper integration of component and coordination codes. Notwithstanding, our experiments demonstrate that we can achieve the similar performance.

For our experiments we chose two applications as testbeds. The first one is a face detection algorithm that processes images in a pipeline. The pipeline consists of three stages, each classifying various features of the face. The application is used to compare various communication patters: processing in a pipeline with traversal of a data/control flow graph. The second testbed is Cholesky decomposition, a linear algebra algorithm, which lends itself easily to parallelization for a multi-core system. It decomposes a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. For such matrices, the algorithm is roughly twice as efficient as the more general LU decomposition. We use a tiled version of the algorithm, originally proposed by Buttari et al. [3]. Our choice is motivated by the fact that the algorithm is well used in computational linear algebra as well as having a sufficiently complex and varied internal structure to benefit from component coordination as a method of concurrent implementation.

Contribution. The main contribution of the paper is conceptual comparison and performance evaluation of two coordination approaches, a type-based coordination language S-Net and a C++ template library Concurrent Collections (CnC) from Intel [4]. The choice is justified by the fact that the models provide different mechanisms for communication of decoupled components. CnC components interact by sharing access to the memory that holds tuples. On the other hand, communication in S-Net is based on message passing in a statically defined communication graph. Furthermore, routing in S-Net heavily relies on types associated with messages. CnC, in contrast, uses the type system of the implementation language (C++, in our case) for error checking and does not rely on typing information at runtime.

CnC is quickly gathering momentum in both industrial and academic research [5, 6]. S-Net, in contrast, challenges CnC with an emphasis on rapid parallel program design. Several industrial use cases have been developed by our partners including Thales Research, SAP and Philips Healthcare. These include signal processing [7] and image processing [8] algorithms. Comparative studies of S-Net against other coordination languages have not yet been conducted; this comparison against CnC is the first.

The measurements that we present below show that on the chosen applications both S-Net and CnC show similar performance, even though the design decisions are different. S-Net provides a cleaner separation of concerns, yet there is a set of overheads associated with message passing. On the other hand, CnC requires a tuning mechanism for efficient execution.

The rest of the paper is structured as follows. In Section 2, we overview existing coordination technologies that are based on tuple space and streaming models. In Section 3, we briefly explain CnC and S-Net coordination models. Section 4 presents evaluation of the face detection algorithm, while Section 5 discusses implementations of the Cholesky decomposition algorithm in both CnC and S-Net. Section 6 discusses advantages and disadvantages of the coordination models based on the evaluation results obtained. Finally, Section 7 concludes the paper.

2. Related Work

Coordination languages promote separation between computational processes and their coordination. In particular, coordination is responsible for process synchronization and communication. If computation and coordination models are separated, the application can be developed in a decontextualized manner [9]. The application can be divided into a set of non-interfering tasks, each of them assigned to a separate domain expert that does not need to be aware of implementation details in other parts of the program. Coordination layer provides a glue that connects the components in a complete application. CnC uses a tuple space model as a glue and S-Net connects the components by means of streams. Various coordination languages and technologies have adopted each of these approaches.

2.1. *Tuple space models*

Linda is often considered the earliest coordination language [10, 11]. Communication between independent processes is performed using shared memory referred to as the tuple space. Occupants of the tuple space are accessed by logical name, therefore, the only information the processes share is the discipline of occupants' tag use. However, the separation of concerns in Linda is not complete, because synchronization has to be coded in the computational part of the program.

Many further languages adapted Linda's tuple space model. Lime [12, 13] provides a framework and middleware for coordination of mobile applications that spans across multiple volatile devices. Hierarchical tuple spaces is an encapsulation mechanism that Lime provides [14]. Thus, a Lime application can be developed by multiple programmers without causing undue interference. NetWorkSpaces [15] has a similar abstraction mechanism that facilitates modularity. A centralized server takes control over multiple workspaces that store tuples and provide a scoped communication mechanism for processes that know the name of the workspace. The purpose of this coordination model is to deliver high performance to languages that are originally aimed for prototyping and rapid development, such as Matlab, Python, Perl and R. Similarly, Swift [16] glues programs written in scripting languages together and executes them in parallel. Instead of tuple spaces, Swift uses single assignment variables as a glue.

2.2. *Streaming models*

Darwin [17, 18] structures components written in various external languages into a hierarchical statically-defined network. The (sub)networks communicate by sending typed messages from output ports to input ports. In the strongly-typed coordination language Manifold [19] components communicate with each other over streams of data. Communication concerns are moved outside the components and are managed by special communication primitives called *manifolds*. The manifold declares an event-driven interaction protocol for the components, which are oblivious to the environment. As a result, the components can be reused in many applications without being modified.

A descendant of Manifold is Reo [20] that follows the same approach to coordination. However, instead of manifolds, Reo has a set of channels that are used as primitive constructs for building communication models in concurrent systems. The main advantage of the primitives is that they can model other communication systems (e.g. the ones based on message passing, shared spaces, remote procedure calls). Furthermore, the channel-based communication model supports anonymous communication and synchronization, where the components need not be aware of the destination of any messages. The interaction between channels is statically defined as a formal model. Reo uses this model to argue about its correctness and properties of communication [21].

3. S-Net and CnC

3.1. *The Parallel Component Technology*

Decomposition and encapsulation are normally regarded as general software engineering strategies, rather than central concepts of parallel computing. Problem decomposition results in the representation of an application as a set of black-box components, whose functionality is defined in terms of the interface description, with some glue code that holds the components together in a way that ensures the expected system behavior.

Generally speaking, not only does a component respond to the input messages by producing and sending its output messages, but it can also have an internal state. When that is the case, messages on the input stream are generally processed by different mathematical functions as they arrive at the input. This prevents messages from being processed out of order and simultaneously with one another. It also prevents the component from being moved from place to place or cloned in a multicore system, since the new copy of the component would have to be created in the same state, which is internal, and hence unavailable. Yet many components would not have a persistent internal state.

A possible solution is to structure and manage state transitions in the component world in the same way as control flow is structured and managed in ordinary programming. A way to achieve this is to strip user-defined components of all per-

sistent state, so that they become pure functions that map a tuple of parameters into a similar collection of results.

As soon as the results are produced, the internal state should effectively be destroyed. Such components are easy to reason about and debug, they are inherently mobile, and usable as a black box in a parallel computing environment — but there is also a price to pay. The gluing environment would have to provide sufficient scaffolding to support an evolving state (or local states!) of the computation. In other words, it will need to hold the effective state of one or more components for them and present it back to the components' inputs in combination with any data to be processed. This is similar to thread-safe code where the intermediate state is held in thread-local memory. Except in this case it is not the intermediate state, but rather, say, the final state of the current iteration that is held and managed outside the component.

3.2. *The* S-Net *Language*

An S-Net application consists of components connected by anonymous data streams [2]. The application is represents as a network between an input and an output, which are two external streams that connect the application with its environment. In the following we briefly review the most relevant concepts of the language.

The box concept. A component is instantiated as a Single-Input, Single-Output (SISO) box. The box has a limited life cycle: it accepts one item from the input stream, does some processing and yields zero, one or more items to the output stream, after which it destroys its internal state and waits for the next input item to arrive. Components are written in a box language; as of today we support C and the functional array language SaC [22] as box languages. An S-Net box is associated with C or SaC function. Towards S-Net that box function is characterized by a type signature that defines the type of records that the box accepts and, in a similar way, the types of any output records that the box may produce.

The streaming data concept. All boxes accept records as input. A record in S-Net is a set of *fields* and *tags*. Both fields and tags have names and values. Field values cannot be examined in S-Net: they are references to data which are private to the box language. Tags are standardized as integers and their values are available in both the box language and S-Net. Streams between boxes are sequences of records. Even though all boxes are SISO, the data relationships between them are not one-to-one, since streams can be split and merged using combinators.

Synchronization. In S-Net the only component which can store and combine state is the *synchrocell*. For example, the expression $[|\{r\}, \{s\}|]$ synchronizes precisely two messages: one whose type contains at least a field r, another with at least s. All other messages remain untouched and are forwarded further. The semantics of a synchrocell is sequential and does not involve any computation in the narrow sense, hence the concurrency and mobility concerns do not apply.

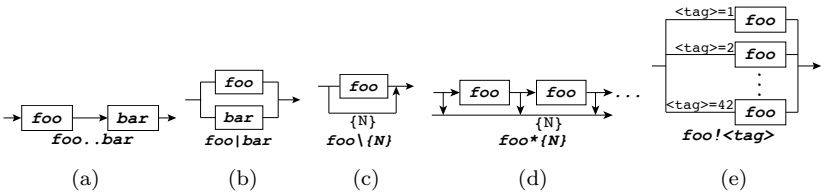


Fig. 1. Illustration of network combinators used in the benchmark applications for S-Net: serial combinator (serial composition) (a), parallel combinator (parallel composition) (b), feedback combinator (feedback loop) (c), star combinator (serial composition) (d), and split combinator (parallel replication) (e)

Combinators. These are second-order functions that support composition of SISO networks, as illustrated in Figure 1. Starting with the simple combinators, we identify serial composition (..), where the output stream of one component becomes the input stream of another component, and parallel composition (|), where records are effectively routed through one of two alternative boxes or subnetworks based on their types.

The feedback combinator (Figure 1(c)) examines the output records of an operand network and redirects those records which match a pattern back into the input of that network. E.g. $C\backslash z$ creates a feedback loop around operand network C for records for as long as they match pattern z. This allows a single operand network C to repeatedly process a record until a different kind of record is produced, carrying the result.

There exist combinators for dynamic replication of a SISO network. The expression $A*x$ will serially replicate the operand network A an unspecified number of times (Figure 1(d)). Only records that match a pattern x escape this network. I.e. the expression is equivalent to an unbounded serial expansion $A..A..A..$ for records as long as they do not match pattern x. Typically, at some point in time, due to processing by network A, they no longer match and will then appear on the output stream of the expression.

Similarly, the expression $B!<y>$ defines a split combinator (Figure 1(e)). It replicates the SISO network B an unspecified number of times in parallel. For every unique tag value y one parallel branch of B is created. This can be seen as an unbounded parallel expansion $B_{y_0}|B_{y_1}|B_{y_2}|\ldots$ The branches are persistent in that records with the same tag value y are sent by the coordinator down the same branch.

Network types affect message routing. When a channel splits using a parallel combinator (Figure 1(b)), the records are sent to the operand network that matches its type best [1]. Type specifications of complex networks are inferred automatically by the S-Net compiler.

Flow inheritance. S-Net supports structural subtyping on the record organization of data messages. In practice this means that a box only knows about the fields and tags that it actually expects (and receives) or produces. At the same time

incoming records may contain any number of further fields and tags. These are routed around the box and attached to any outgoing record produced by the box in response to that incoming record. Flow inheritance significantly improves flexibility in network construction and reuse opportunities for boxes in different network and application contexts.

3.3. *The Concurrent Collections Model*

A detailed description of the Concurrent Collections model can be found in [23]. In the following, we merely present a brief description of the CnC concepts relevant to our case studies.

Following the coordination approach, the CnC model decouples computation from the expression of its parallelism. Consequently, a domain expert determines the design of the algorithm, and a tuning expert can be called upon to deal with parallelism, communication, scheduling and distribution issues.

The domain expert specifies the computation as a graph with the following types of nodes.

- **item:** A collection of instances of a data *item* of a certain type. Item-instance collections are used to represent data. Each instance is a unit of storage, communication, and/or synchronization, which is distinguished within a collection by a tag, see below.
- **step:** A collection of instances of a computational *step*. A step is an indivisible unit of sequential execution specified by the domain expert, which can be instanced (i.e. cloned and run) on specific item instances. An instance of a step may read items from statically known item collections and may place new items in the same or different statically known item collections.
- **tag:** A collection of *tag* instances. Each instance of a step has a unique control-tag instance and each item instance has a unique data-tag instance. A tag instance is an instance of a tag, which is a tuple of *tag components*. The presence of a control tag instance indicates that an instance of a step will execute at some point in time, but it does not tell *when* this may happen. A step may produce tag instances as well. A step-instance collection is statically associated with exactly one tag-instance collection, which *prescribes* the step. The tag-instance is an input to the step program alongside all the item-instances the step-instance may read.

A step has a fixed structure:

- **input section:** This is the opening part of the step program where the step reads item-instances from its statically known input collections after computing the corresponding data-tag instances based on its control-tag instance and after sending them to the runtime system to gain access to the actual data.

- **pure function:** This represents side-effect-free computations that work out the step's results based solely on the control-tag instance and the item-instances read. No data is written to any collection at this stage.
- **output section:** The section optionally places the results from the previous section in the form of newly computed item-instances into the statically known output collections, and also optionally places new control-tag instances in statically known control-tag collections.

The first two sections can be intermingled when input requires some tag computation, possibly involving further item-instances.

A step instance begins its life when a control-tag instance is placed in the corresponding control-tag collection by another step-instance. At this point it is said to be in the *prescribed* state. As soon as all the input item-instances become available, the step-instance is in the *enabled* state. All enabled step-instances will eventually be executed, but it is up to the runtime system to determine when.

Even though the relations between step, data and tag collections, which define the communication graph, are available statically, this information is not sufficient to achieve optimal scheduling and resource management. As a remedy CnC provides additional *tuning* mechanisms. The problem here is that an instance of a step may depend on a-priori unknown item-instances from the statically known collections. In order to determine which item-instance those are, the runtime system would have to run the step. If the item-instances are not yet available, the step-instance would stall and need to be re-launched later. This wastes resources, but does not jeopardize the semantics thanks to the purity of the middle section of the step. To remedy this inefficiency, CnC permits the programmer to supply a *dependency function* which can be called by the runtime system to determine which item-instances the current control-tag instance will cause the step to fetch. If those are available from the corresponding collections, the step-instance is judged to be enabled straight away and is run whenever appropriate without the risk of a roll-back.

In S-Net, synchronization is performed by synchrocells, occurring in a coordination program. This obviates any external mechanism, such as the dependency function. In CnC terms, an S-Net step-instance is instantly enabled by an input message, which always comes pre-synchronized from the coordinator.

4. Case Study: Face Detection

Our first benchmark is a face detection algorithm [24]. It is a small yet very practical application, which illustrates conceptual and runtime differences between the coordination languages. A simplified version of the algorithm, which simulates the detection by abstracting images as strings, has already been implemented in CnC. Based on the existing prototype, we implemented complete applications for both platforms.

The application consists of three feature classifiers connected in a pipeline, each detecting one of: a face outline, eyes and a mouth. If one of the classifiers fails to find

the corresponding feature, the image is judged to not include a face. The result of classification is forwarded to an accumulator that collects all processed images and returns the rate of images that contain a face. The feature detection is performed by calling `cvHaarDetectObjects` function from the OpenCV [25] image processing library.

A pipeline is the most common communication pattern in S-Net. We use this application to compare S-Net pipelined computations with a CnC computation model driven by traversal of a data/control flow graph.

4.1. *Implementation in Intel Concurrent Collections*

The computational graph of the CnC implementation is given in Figure 2. Before execution, the environment adds images to a item collection `image` and tags that represent image indexes to a tag collection `classifier1_tags`. Once added, the images are never removed from the item collection. Therefore, the item-instances are always available, and computations are essentially control-driven and prescribed solely by the corresponding tag collections. For every tag in `classifier1_tags`, the runtime executes a component `classifier1`, which attempts to find a face outline in the image. If it succeeds, a relevant tag is added to a tag collection `classifier2_tags`, which contains a subset of tags from `classifier1_tags`. The classifier `classifier2` attempts to find eyes in the image. If the eyes are found, the classifier adds corresponding tags to a tag collection `classifier3_tags`. A classifier `classifier3` is the final stage of the pipeline, which detects a mouth in the image. It stores computation result in a data collection `face`, which contains indexes of the images that are judged to contain faces. The collection can be accessed by the environment after the CnC application terminates.

In this application all item instances in the `face` collection are available before step execution. Thus, dependency functions are not required for efficient scheduling.

4.2. *Implementation in* S-Net

The face detection algorithm in S-Net is implemented as a pipeline that consists of three classifiers and a subnetwork `Collect`, which collects classification results from the classifiers and calculates the number of images that contain faces (Figure 3). Each classifier receives a record containing an image identifier `name` and an image `img` in an OpenCV format. All classifiers, apart from the last one, have two logical output streams. If the classifier detects a face feature, it sends the input message to the next classifier. Otherwise, it sends the classification result `<isFound>` = `false` and the image identifier `name` to the subnetwork `Collect`. As a result, the image bypasses the remaining classifiers if one of the features has not been detected.

The `Collect` subnetwork receives input messages of two types. Messages of the type `(<isFound>, name)` are produced by the classifiers as described above; messages of the type `(<N>, <rate>)` store a state of the `Collect` subnetwork. Tag `<N>` is an integer that stores the number of images that has still to be processed

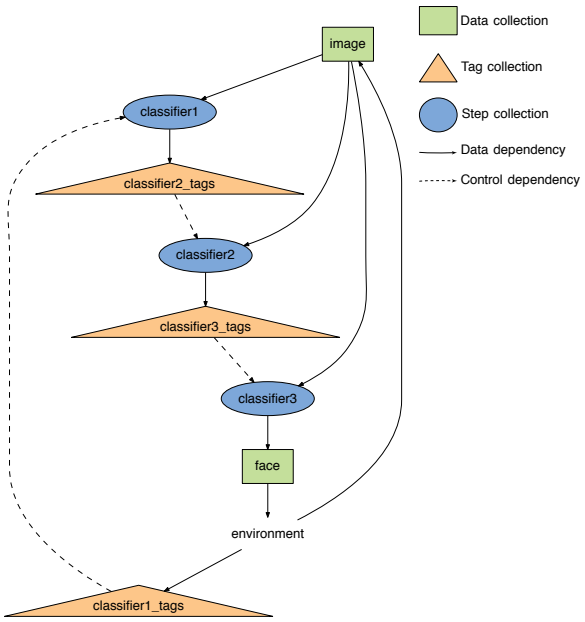


Fig. 2. A computational graph of the CnC implementation of the face detection algorithm.

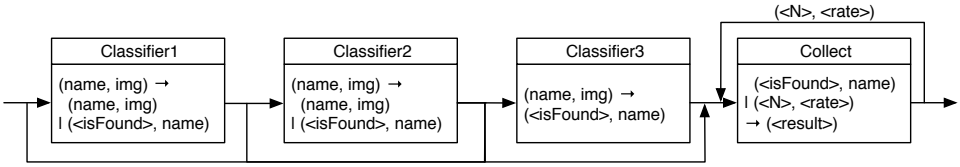


Fig. 3. The S-Net network for the face detection algorithm.

and `<rate>` is the number of images that contain faces. If `<N>` $= 0$, then all images have been processed and a program produces a message `(<result>)` that stores the number of images with faces.

In order to gather information about processed images, the `Collect` subnetwork contains a synchrocell that combines messages of types `(<N>, <rate>)` and `(<isFound>, name>)` and then generates a state record of type `(<N>, <rate>)`.

Synchronization in CnC and S-Net is implemented in two different ways. The natural way to perform synchronization in CnC is to gather elements in a data collection and then process them in a single step instance. In contrast, synchrocells in S-Net can synchronize only two messages at a time, and, therefore, synchronization of multiple messages has to be implemented as a component with reduction semantics. An advantage of this approach is that it does not require *all* messages to have arrived to start synchronization.

Vendor	AMD
Processor Model	Opteron 6174
Processor Name	Magny-Cours
Clock(GHz)	2.2
# Sockets	4
Cores(Threads)/Socket	12(12)
L1 Data Cache	64 KB/core
L2 (Data and Instruction) Cache	512 KB/core
Shared L3 Cache	12 MB
DRAM Capacity	256 GB
Operating System	Scientific Linux 7.1
Kernel Version	3.10.0-229.1.2.el7
Compiler	GNU GCC 4.6.3

Fig. 4. Evaluation platform for experiments

4.3. *Experimental Evaluation*

In addition to implementations in CnC and S-Net, we implemented three versions of the face detection algorithm. The first one is purely sequential and executes the detection cascade for each image. The detection cascade is implemented as classifying functions that are called in a sequence.

The second one is a parallel version obtained by adding OpenMP annotations, which specify that processing of all images can be run in parallel, to the first version. The pipeline itself is sequential. In seems that despite the ongoing research on pipeline parallelism in OpenMP [26], exploiting the parallelism using the standard OpenMP is currently nearly impossible.

Finally, the third version is implemented using PThreads: images are put in a job queue and are retrieved by available workers. In these implementations, images that need to be processed are equally distributed among the cores. Stages of the pipeline are run sequentially. Although pipeline parallelism can be implemented in PThreads, this would require a lot of work. We use OpenMP and PThreads conventional implementations as yardsticks for the CnC and S-Net ones.

The CnC model permits many runtime system designs, including those for distributed memory using MPI. We use Intel CnC 0.9, which is a shared-memory implementation that uses Intel Threading Building Blocks as a threading layer. Step collections are implemented in C++.

Boxes for S-Net are implemented in C. All S-Net experiments make use of the Front runtime system [27], which combines very low overhead of S-Net network maintenance with efficient transportation of records throughout the network.

All five implementations are compiled using GNU GCC 4.6.3. Our test machine is a 4-socket system with 12-core AMD Opteron 6174 processors and a total memory capacity of 256 GB RAM. Fig. 4 provides further details.

There are two main parameters that affect the performance of the face detection application: the size of an individual image and the number of processed images. The former affects the amount of work done by individual components and the latter influences the total application throughput. Figure 5 and 6 demonstrate the

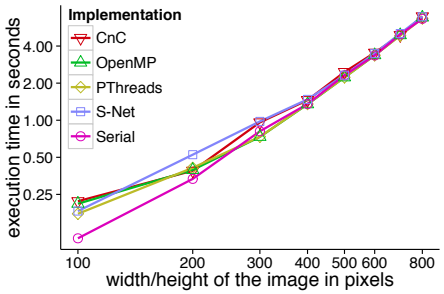


Fig. 5. Execution time of the face detection algorithm for various image sizes

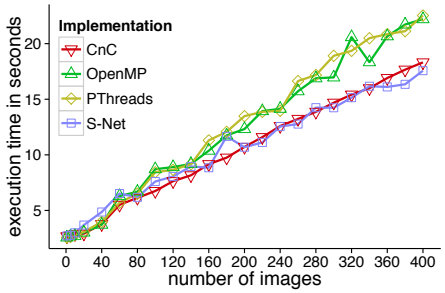


Fig. 6. Execution time of the face detection algorithm for various numbers of images. Each image is 500 px × 500 px; a bigger batch of input images contains smaller ones as subsets

performance impact of these parameters. The experiments were performed using 48 parallel threads.

In Figure 5 one image of various sizes is processed. We use this experiment to measure the overhead in each implementation. The serial version, which is taken as a baseline, unsurprisingly processes the smallest images faster than other implementations. The rest of them spend 60–110 ms (36%–50%) on thread management. When an application processes images of larger sizes, the overhead is dominated by the image processing time. For our next measurement we decided to use images of size 500 px × 500 px for which the thread management overhead is negligible.

Figure 6 demonstrates the throughput achieved by each of the five implementations. When the number of images increases, CnC and S-Net show better performance compared to parallel implementations in PThreads and OpenMP. This is achieved by using work-stealing execution strategy that dynamically allocates jobs to workers that are not busy.

In Figure 7 we visualize the runtime execution for an input that consists of 400 images with 500 px × 500 px each. In Figures 7(c) and 7(d) there is much white space on the right, meaning that at the end of the run-time the tasks are not scheduled in an optimal way. In PThreads and OpenMP implementations tasks are uniformly distributed between threads before the image processing begins. However, due to different processing times of individual images, there is load imbalance at the end. In contrast, scheduling in CnC and S-Net is based on a work-stealing strategy: when a worker runs out of work, it "steals" tasks from the queues of other workers.

White space on the left of Figure 7(b) illustrates that S-Net spends a fraction of execution time just warming up: it is either busy scheduling new jobs or the workers are not being fast enough to fetch jobs from their queues.

The figures also illustrate that stages of the classification cascade in CnC and S-Net are executed in different orders. A CnC thread executes all stages for a single image before starting processing the next image. S-Net, in contrast, executes the first stage of the pipeline for all images and only then executes the second and

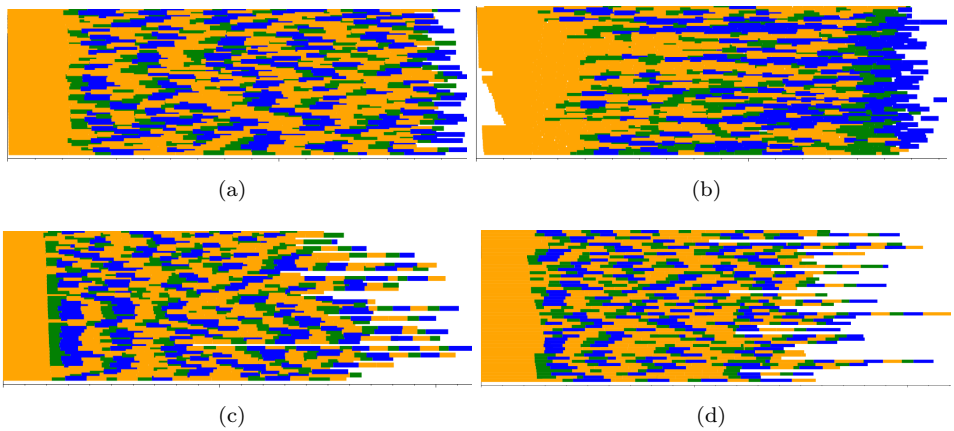


Fig. 7. Visualization of the face detection application runtime execution for implementation in CnC (a), S-Net (b), PThreads (c) and OpenMP (d). The X axis is time, the Y axis is an execution thread, orange is the first (face outline) classifier, green is the second (eyes) classifier, blue is the third (mouth) classifier

the third one (visually, the orange color, which represents execution of the first classifier, dominates on the left of Figure 7(b); the green color, which represents execution of the second classifier, dominates in the middle; and blue color, which represents execution of the third classifier, dominates on the right). The Front runtime system aims to execute box instances that contain the greatest numbers of messages in their input buffers. Therefore, initially S-Net prioritizes the first classifier that contains 400 images in the input queue. Then, after the first classifier has processed the majority of input messages, the runtime system executes the second and the third stages.

Implementation of the face detection algorithm using various models demonstrates that applications with non-uniform message processing time require an execution strategy that monitors load and dynamically reschedules tasks. As a solution, CnC and S-Net provide the work-stealing execution strategy.

5. Case Study: Cholesky Decomposition

Our second case study is a numerical algorithm from the linear algebra domain: Cholesky decomposition. It solves the following problem: given a symmetric positive definite matrix A, find a lower-triangular matrix L, such that $A = LL^T$. We use the tiled version of the Cholesky decomposition algorithm described by Buttari et al. [3].

Initially, the input matrix A is decomposed into $p \times p$ blocks A_{ij} of size $b \times b$ each. The Cholesky decomposition problem is first solved for all blocks of the 0th column of the block matrix A_{i0} separately. Next, all blocks in the low-triangular block-submatrix A_{ij}, $i \in [1, p-1], j \in [1, i]$, are recomputed and the algorithm

```
1: for k = 0, ..., p − 1 do
2:   L_kk ← InitialFactorization(A_kk)
3:   for all j = k + 1, ..., p − 1 do
4:     L_jk ← TriangularSolve(L_kk, A_jk)
5:   end for
6:   for all j = k + 1, ..., p − 1 do
7:     for all i = k + 1, ..., j do
8:       A_ij ← SymmetricRankUpdate(L_jk, L_ik)
9:     end for
10:  end for
11: end for
```

Fig. 8. Cholesky decomposition tiled algorithm

recurses into it.

The complete algorithm is given in Figure 8. The computations form a series of iterations with the iteration index k, each divided into three steps:

Initial Factorization. A scalar Cholesky decomposition algorithm is used to solve $A_{kk} = L_{kk}L_{kk}^T$ equation on this stage. The result of computation is a lower-triangular block L_{kk}, where L_{ij} is a block-matrix partitioned in the same way as A.

Triangular Solve. During this phase we apply the result of the previous step to solve the equation $A_{jk} = L_{jk}L_{kk}^T$. The result is the block L_{jk}. This step can be performed for all blocks in the same column concurrently.

Symmetric Rank Update. This step is used to update values of blocks A_{ij}, where $j \in [k+1, p-1]$ and $i \in [k+1, j]$. This is done using the formula $A'_{ij} = A_{ij} - L_{ik}L_{jk}^T$. Similar to the previous step, this can be done concurrently for all i and j.

The problem thus boils down to the above component algorithms. Our job is to glue them together using a coordination program. The Cholesky decomposition algorithm is particularly challenging for parallel implementation, as it gives rise to a varying number of parallel threads.

5.1. *Implementation in Intel Concurrent Collections*

In the following we briefly sketch out the CnC implementation of tiled Cholesky decomposition published in [28].

Figure 9 presents the data/control flow graph of the Cholesky decomposition algorithm in CnC. Item collections `p` and `b` store the total number of blocks in the initial matrix and the block size, respectively. The input matrix A and output matrix L are both stored in the item collection `Lkji` and are accessed from the step collections. The component algorithms are implemented as steps in the implementation: `InitialFactorization`, `TriangularSolve` and `SymmetricRankUpdate`. Their behavior is defined by the data collections `Lkji`, `b` and `p` that they use as

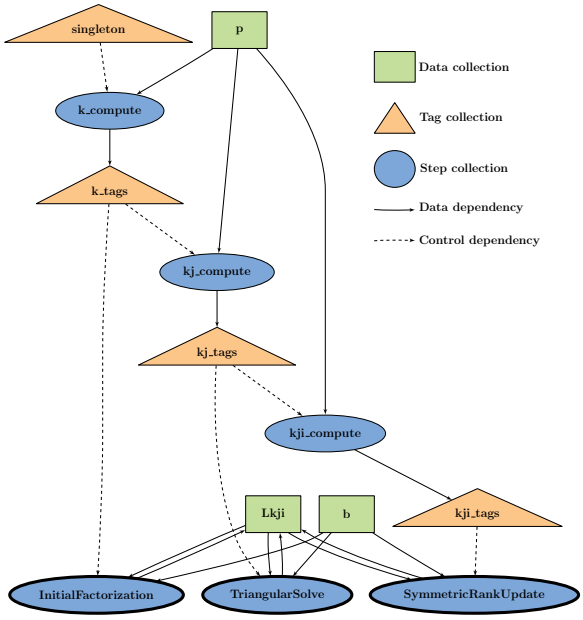


Fig. 9. A computational graph of the CnC implementation of the Cholesky decomposition.

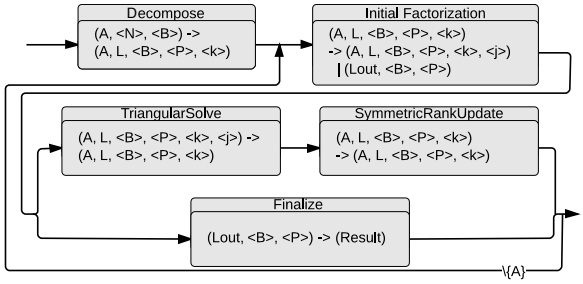
input. Tag collections `k_tags`, `kj_tags` and `kji_tags` control the behavior of each step collection. All of these steps produce the result of the computation and put it back into the item collection `Lkji`. Furthermore, there are three auxiliary step collections (`k_compute`, `kj_compute` and `kji_compute`) that generate tags for tag collections and prescribe execution of the component algorithms.

Figure 9 contains tags and items that do not have inbound edges. This means that they are taken from the *environment*, which is the external code that invokes the computation. For our example the environment provides item collections `Lkji`, `b` and `p` and the tag collection `singleton`.

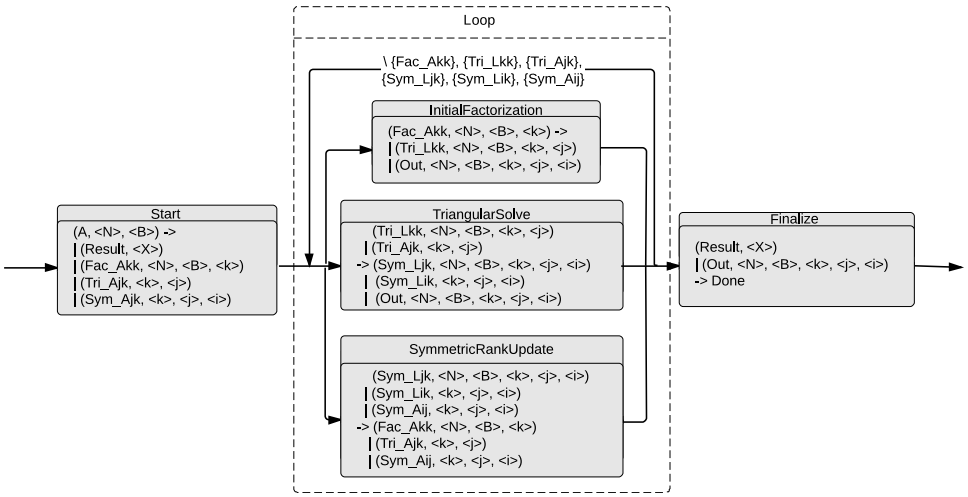
Dependency functions as a *tuning* mechanism allows the programmer to provide information about step data dependencies that is not present in the flow graph. In order to evaluate the effect of the CnC dependency functions, we provide two versions of the implementation for CnC. The first one does not rely on the tuning, whereas the second one uses the dependency functions to improve scheduling. For a fair comparison of S-Net and CnC scheduling algorithms, the second S-Net implementation must be preferred, because the S-Net network already contains the information provided by the CnC dependency functions.

5.2. *Direct Implementation in* S-Net

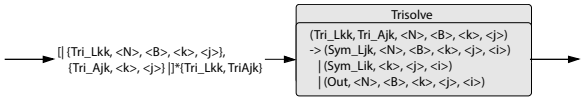
We developed two variants of tiled Cholesky decomposition in S-Net. The first one directly implements the algorithm in Figure 8.



(a) The implementation with barrier synchronization



(b) The data-driven implementation



(c) The structure of the `TriangularSolve` subnetwork in the data-driven implementation

Fig. 10. The S-Net networks of the Cholesky decomposition algorithm

Unlike CnC, the coordination network in S-Net does not specify a control flow. Box computations only depend on the availability of input data. Data relations are completely defined statically. Consequently, the consistency of a coordination program can be verified at compile time. Again, unlike the CnC graph, the S-Net network is hierarchically structured.

In Figure 10(a) we illustrate the S-Net network for the Cholesky decomposition algorithm. A (coordination-level) type signature is displayed with each box. `TriangularSolve`, `SymmetricRankUpdate` and `Finalize` are subnetworks that contain boxes inside.

The `Decompose` box receives a message with the input matrix A, its size N and the block size B from the environment (tag values enclosed in angular brackets are "analyzable" by the compiler and runtime system). It reallocates the array where the input matrix is stored and permutes the matrix elements there in order to improve spatial and temporal locality. In addition we add an index $k = 0$ to the output message. As a result, the `Decompose` box outputs the input matrix with permuted elements A, the output matrix L filled with zeros (on each iteration of the feedback loop combinator we add new values to the matrix), the block size B, the number of blocks P and an additional index k.

Next, we perform recursive computations (the outer loop in Figure 8). The recursion is expressed using a feedback loop combinator. The combinator redirects the output of the `SymmetricRankUpdate` box to the input of `InitialFactorization` box as long as the message containing a field of type `A` is produced by the `Finalize` box. If so, the feedback loop stops and the message is sent to the output stream.

The `InitialFactorization` box performs the first step of the algorithm in Figure 8. Also we compare the loop index k with the number P in order to determine whether all the blocks have been processed. Depending on the result, messages of different types are sent. If k is still less than P, we add computed elements to the matrix L and supply the input record with the additional index j. The `TriangularSolve` and `SymmetricRankUpdate` boxes perform the computations of the corresponding stages. Lastly, the `Finalize` box winds up the computation by converting the result matrix to a format suitable for output and then deallocates memory.

5.3. *Data-Driven Implementation in* S-Net

The drawback of the direct S-Net implementation of the tiled Cholesky decomposition algorithm is the barrier synchronization needed after each iteration. This is not only costly in general, but it also disadvantages S-Net compared with the CnC implementation sketched out before that carefully avoids these barriers. Therefore, in the following we devise an entirely data-flow driven S-Net implementation that eliminates the performance disadvantage of the direct implementation. Figure 10(b) illustrates our second approach.

The box `Start` receives the same input, consisting of matrix A, matrix size N and the block size B. It produces the record containing the matrix `Result` filled with zeros and number X — the number of blocks with the result needed to construct the output matrix. This message is propagated forward to the box `Finalize` using the flow inheritance mechanism sketched out in Section 3.2: if the type of a message is not matched by the box type scheme, the runtime system bypasses this message forward until it is accepted by a box. In addition, the box `Start` produces initial messages containing separate blocks corresponding to a stage in the algorithm. Thus, it produces messages with diagonal blocks stored in `Fac_Akk` and accepted by the `InitialFactorization` box; messages with blocks from the column j stored in

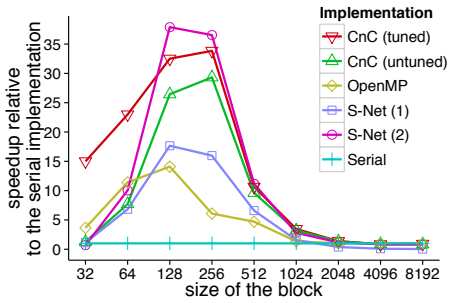


Fig. 11. The speedup relative to the serial implementation of Cholesky decomposition CnC and S-Net applications on 48-core machine for input matrix of size $N = 8192$

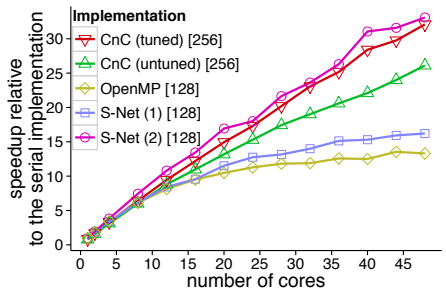


Fig. 12. The speedup of Cholesky decomposition CnC and S-Net applications for different number of cores (for matrix of size 8192×8192 and optimal block size, which is specified in brackets)

`Tri_Ajk` and accepted by `TriangularSolve`; and messages corresponding to column j and row i stored in `Sym_Aij` and accepted by `SymmetricRankUpdate`.

Messages are transfered between three boxes mentioned above until the blocks with the final value are produced and accumulated by the `Finalize` box. Subnetworks `TriangularSolve` and `SymmetricRankUpdate` contain a synchrocell inside that gathers data dependencies of boxes before executing them. The synchrocell in the first subnetwork awaits two messages with `Tri_Ajk` and `Tri_Lkk`, representing blocks A_{jk} and L_{kk} respectively (see Figure 10(c)). Similarly, synchrocells in the second subnetwork waits for three blocks L_{jk}, L_{ik} and A_{ij} packed into `Sym_Ljk`, `Sym_Lik` and `Sym_Aij`, respectively. Once the dependencies are satisfied, the box begins to execute and will eventually produce output messages (or decide not to). The block with the final result is packed into the `Out` field — the message containing it is delivered directly to the last box `Finalize`.

Such a network structure avoids global synchronization after each step. Messages do not block each other and computations start as soon as all *local* dependencies are satisfied.

5.4. *Experimental Evaluation*

Cholesky factorization is a significant example of a computational linear algebra problem. It is affected by locality (block size), parallelism (the number of cores used) and data dependencies between blocks. The amount of work and available parallelism varies during program execution time. This challenges both systems' abilities to manage the resources of a concurrent platform and exposes their differences in doing so. For all measurements presented in this section we used the same evaluation platform as for the face detection case study (see Figure 4 for details).

We provide the evaluation results for two CnC, two S-Net and one OpenMP implementations, the last one merely intended as a basis for comparison. Figure 11

shows the speedup referenced to the serial implementation vs the block size. The serial implementation represents the tiled Cholesky decomposition algorithm for a single core. It is similar to the implementation in CnC, but lacks the coordination overhead. The figure shows that the performance depends on the size of a block and the number of blocks in the partition. The peaks in the figure demonstrate optimal values for the block size.

The OpenMP implementation illustrates what performance can be achieved if a domain expert without adequate experience in parallel computing attempts to parallelize a serial version of the application. We added OpenMP annotations to the serial implementation that specify independent iterations in loops in lines 3 and 7 of Algorithm 8. The major disadvantage that causes performance loss in this implementation is global synchronization barrier between any two loop iterations.

In other implementations, each block represents a separate message or a tuple, so blocks are explicitly independent. This solves the latter issue. However, the direct S-Net implementation called S-Net (1) still suffers from global barrier synchronization. Both CnC (CnC (untuned) and CnC (tuned)) and the data-driven S-Net (S-Net (2)) implementations overcome this by specifying computations as a dependency graph. Hence, independent computations are not unnecessarily synchronized and every block in the tiled Cholesky decomposition algorithm can be computed as soon as all other blocks it depends on have been completed.

Iterations of the outer loop (line 1 in Algorithm 8) have to be executed in order. Cyclic dependencies between iterations reduce parallelism in this application. Figure 13 demonstrates a data dependency graph between iterations (an iteration is defined by the tuple (k, j, i)) for a 4×4 block-matrix as the input. The figure shows a partial order, with the maximum number of parallel threads dynamically varying from iteration to iteration. Scheduling must observe the partial order and should give priority to execution steps that are on the critical path.

The S-Net runtime system Front attempts to process records that have advanced the furthest in the processing graph. Here, workers generally favor records that come to a box with an empty output stream. Consider the example in Figure 13. Assume that a worker needs to select a record for processing from the choice as defined by tuples $(0, 2, 1)$, $(0, 2, 2)$ and $(1, X, X)$. Front would select the tuple $(1, X, X)$, which represents a critical dependency whose elimination potentially increases the available parallelism [27].

Threading Building Blocks controls execution of the CnC program. Similarly to Front, it uses a work-stealing scheduler to schedule tasks [4], which is particularly efficient in an application with dynamically varying concurrency levels.

The use of dependency functions in CnC (tuned) brings significant improvement compared to the CnC (untuned) version of the program, especially when there is much exploitable concurrency. For the best performance range of block size the improvement caused by dependency functions is about 15%. On the other hand, an overhead introduced by the dependency functions for cases with a small amount of concurrency caused a 7% loss in performance.

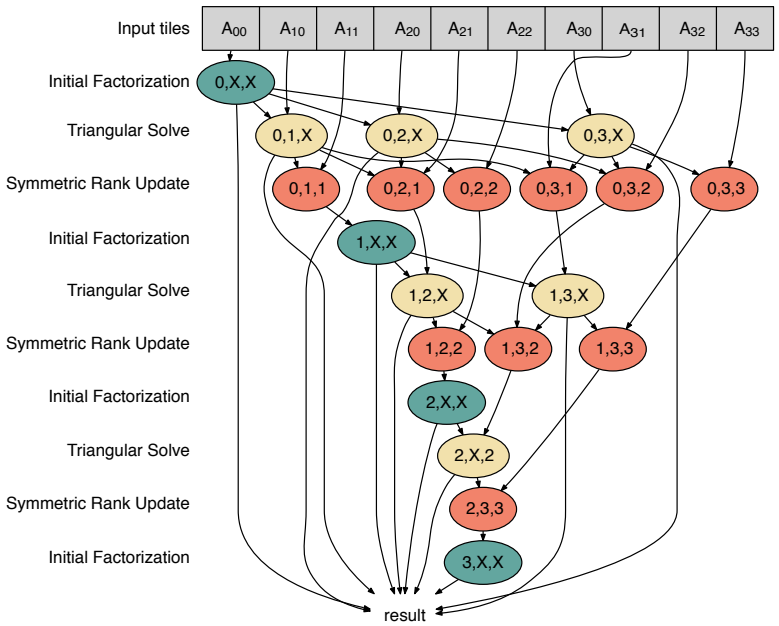


Fig. 13. An example of the data dependency graph for Cholesky decomposition ($p = 4$). The data item is defined by a step (each step is represented by a different color) and a tuple (k, j, i). X specifies that the value is not defined

Figure 12 shows the speedups of S-Net and CnC for varying numbers of threads/cores used. Both CnC implementations as well as the data-driven S-Net (2) implementation demonstrate almost linear speedup. In contrast, the S-Net (1) implementation as well as OpenMP implementation suffer from an extra overhead caused by the synchronization barriers.

A complete performance evaluation of the tiled Cholesky decomposition in CnC can be found in [28]. The measurements illustrate that for efficient execution of this application a barrier synchronization needs to be avoided and a data-driven approach is preferred. The runtime must perform dynamic load balancing of tasks to avoid stalls during execution. The load balancing in S-Net and CnC is based on work-stealing, which proves quite efficient in practice. S-Net on this example has a performance similar to CnC and both coordination programming models are effective instruments for implementing applications for multi-core platforms.

6. Discussion

In terms of expressiveness, all CnC and S-Net implementations are well-structured, and there is a reasonably low amount of extra code required for coordination. The coordination code in S-Net includes a description of the topology that connects processing boxes using a language of combinators. Separation between coordination and computation is achieved because coordination relies only on components'

interfaces while box implementation details are left to domain experts.

Strict separation of concerns between coordination and computation in CnC is not complete. Synchronization of input data takes place inside components, so a domain expert needs to be aware of component interaction while designing individual components.

An extra tuning step in the form of dependency functions is required for efficient scheduling in CnC. These dependency functions yield a refinement of the dependency graph for the runtime system. The dependency functions allow a runtime system to predict not only which data collections will be referenced, but exactly which data items in them will be requested by some component instance so that those items may be pre-synchronized, i.e. ensured to be available collectively before running the component. Such kind of tuning mechanism is not needed for S-Net. S-Net components do not deal with synchronization at all: data always comes in pre-synchronized and box execution is exclusively triggered by availability of data.

An application designed for S-Net needs to be structured as a stream processing program. This is the main challenge since not all applications can be easily represented in this way. Such an approach, however, provides a good scalability (use of shared memory is avoided), which is particularly useful for distributed applications [29]. The Cholesky decomposition case study also shows that the composition of computational boxes significantly influences the performance.

To summarize, CnC and S-Net are both systems for coordination of components. Components in CnC are linked by data and control relations. The execution strategy and order are determined mainly dynamically. S-Net uses the message-driven strategy. Components are linked by data relations only. Additionally S-Net offers a hierarchical structuring mechanism and a compile time analysis that follows the structure; most of that analysis can be done statically.

7. Summary and Conclusion

We have compared both expressiveness and performance of S-Net and CnC based on two case studies from the domains of image processing and linear algebra. Whereas CnC can be characterized as a coordination library/specification model, S-Net is a fully-fledged coordination language that achieves near-complete separation of concerns between computing and coordination layers.

We observe that a static network topology and data relations facilitates S-Net compilation and runtime scheduling and communication. S-Net does not use control flow, allowing components to be triggered exclusively by the availability of their input data. Despite the lack of tag collections that determine the sequencing of processing steps, pure dataflow works quite well. S-Net supports a clean separation of concerns between coordination and computation: all data required by a computational component are delivered to it by the coordinator in a single, pre-synchronized message. By contrast, in CnC components must be fully aware of the item collections they access.

Tuning is a feature of CnC that is clearly separated from application design. By introducing dependency functions to the application we evaluated the improvement this may yield. At least in our experiments the improvement was rather small.

We compared the two coordination models with more conventional sequential C and multithreaded OpenMP implementations. With S-Net we managed to achieve excellent utilization of the computing resources provided by our test system without any platform-specific tuning and optimization. The data-driven implementation in S-Net is based on precisely the same sequence of algorithmic steps as the CnC one.

One would think that further improvements for CnC would necessitate a more refined tuning mechanism, while for S-Net further improvements would necessitate better run-time heuristics of stream management.

References

[1] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[2] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010.

[3] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[4] Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the Concurrent Collections programming model. In *CPC09: 14th International Workshop on Compilers for Parallel Computers*, 2009.

[5] Robert Pavel, Robert Bird, Pascal Grosset, Ken Czuprynski, Andrew Reisner, Erin Carrier, Christoph Junghans, Benjamin Bergen, and Allen McPherson. Adaptive mesh refinement under the Concurrent Collections programming model. In *The Sixth Annual Concurrent Collections Workshop (CnC-2014)*, 2014.

[6] Kristin Tufte, Basem Elazzabi, Veronika Megler, Morgan Harvey, Kath Knobe, and David Maier. Using CnC to enable re-use in the Portland Observatory. In *The Sixth Annual Concurrent Collections Workshop (CnC-2014)*, 2014.

[7] Frank Penczek, Stephan Herhut, Clemens Grelck, Sven-Bodo Scholz, Alex Shafarenko, Rémi Barrère, and Eric Lenormand. Parallel signal processing with S-Net. *Procedia Computer Science*, 1(1):2085–2094, 2010.

[8] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Coordinating data parallel SAC programs with S-Net. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[9] Edsger W Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[10] Sudhir Ahuja, N Curriero, and David Gelernter. Linda and friends. *Computer*, 19(8), 1986.

[11] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992.

[12] Gian Pietro Picco, Amy L Murphy, and Gruia-Catalin Roman. LIME: Linda meets mobility. In *Proceedings of the 21st international conference on Software engineering*, pages 368–377. ACM, 1999.

[13] Amy L Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. LIME: A coordination

model and middleware supporting mobility of hosts and agents. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(3):279–328, 2006.

[14] Nicholas Carriero, David Gelernter, and Lenore Zuck. Bauhaus Linda. In *Object-based models and languages for concurrent systems*, pages 66–76. Springer, 1995.

[15] Robert D Bjornson, Nicholas J Carriero, Martin H Schultz, Patrick M Shields, and Stephen B Weston. Networkspace: a coordination system for high-productivity environments. *International journal of parallel programming*, 37(1):106–125, 2009.

[16] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[17] Jeff Magee, Jeff Kramer, and Naranker Dulay. Darwin/mp: An environment for parallel and distributed programming. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume 2, pages 337–346. IEEE, 1993.

[18] Matthias Radestock and Susan Eisenbach. Semantics of a higher-order coordination language. In *Coordination Languages and Models*, pages 339–356. Springer, 1996.

[19] Farhad Arbab, Ivan Herman, and Pål Spilling. An overview of Manifold and its implementation. *Concurrency: practice and experience*, 5(1):23–70, 1993.

[20] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14(03):329–366, 2004.

[21] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in Reo by constraint automata. *Science of computer programming*, 61(2):75–113, 2006.

[22] Clemens Grelck and Sven-Bodo Scholz. SAC — a functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[23] Zoran Budimlic, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sagnak Tasırlar. The concurrent collections programming model.

[24] Intel Corporation. The eight basic design patterns of Intel Concurrent Collections for C++. 2010.

[25] Gary Bradski et al. The OpenCV library. *Doctor Dobbs Journal*, 25(11):120–126, 2000.

[26] Antoniu Pop and Albert Cohen. A stream-computing extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, pages 5–14. ACM, 2011.

[27] Bert Gijsbers and Clemens Grelck. An efficient scalable runtime system for macro data flow processing using S-Net. *International Journal of Parallel Programming*, 42(6):988–1011, 2014.

[28] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc. Performance evaluation of Concurrent Collections on high-performance multicore computing systems. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[29] Clemens Grelck, Jukka Julku, and Frank Penczek. Distributed S-Net: Cluster and grid computing without the hassle. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 410–418. IEEE, 2012.