Distributed Stream Processing with DUP

Kai Christian Bader[1], Tilo Eißler[1], Nathan Evans[1], Chris GauthierDickey[2], Christian Grothoff[1], Krista Grothoff[1], Jeff Keene[2], Harald Meier[1], Craig Ritzdorf[2], Matthew J. Rutherford[2]

[1] Faculty of Informatics
Technische Universität München
[2] Department of Computer Science
University of Denver

Abstract. This paper introduces the DUP System, a simple framework for parallel stream processing. The DUP System enables developers to compose applications from stages written in almost any programming language and to run distributed streaming applications across all POSIX-compatible platforms. Parallel applications written with the DUP System do not suffer from many of the problems that exist in traditional parallel languages. The DUP System includes a range of simple stages that serve as general-purpose building blocks for larger applications. This work describes the DUP Assembly language, the DUP architecture and some of the stages included in the DUP run-time library. We then present our experiences with parallelizing and distributing the ARB project, a package of tools for RNA/DNA sequence database handling and analysis.

Keywords: coordination language, parallel programming, productivity

1 Introduction

The widespread adoption of multi-core processors and the commoditization of specialized co-processors like GPUs [1] and SPUs [2] requires the development of tools and techniques that enable non-specialists to create sophisticated programs that leverage the hardware at their disposal. Mainstream and productive development cannot rely on teams of domain and hardware experts using specialized languages and hand-optimized code, though this style of development will remain applicable to high-performance computing (HPC) applications that demand ultimate performance.

This paper introduces the DUP System[1], a language system which facilitates productive parallel programming for stream processing on POSIX platforms. It is not the goal of the DUP System to provide ultimate performance; we are instead willing to sacrifice *some* performance gain for significant benefits in terms of programmer productivity. By providing useful and intuitive abstractions, the DUP System enables programmers without experience in parallel programming

[1] `Available at http://dupsystem.org/`

or networking to develop correct parallel and distributed applications and obtain speed-ups from parallelization.

The key idea behind the DUP System is the multi-stream pipeline programming paradigm and the separation of multi-stream pipeline specification and execution from the language(s) used for the main computation. Multi-stream pipelines are a generalization of UNIX pipelines. However, unlike UNIX pipelines, which are composed of processes which read from at most one input stream and write to a single output stream (and possibly an error stream), multi-stream pipelines are composed of processes that can read from any number of input streams and write to any number of output streams. In the remainder of this document, we will use the term "stage" for individual processes in a multi-stream pipeline. Note that UNIX users — even those with only rudimentary programming experience — can usually write correct UNIX pipelines which are actually parallel programs. By generalizing UNIX pipelines to multi-stream pipelines, we eliminate the main restriction of the UNIX pipeline paradigm — namely, the inherently linear data flow.

In order to support the developer in the use of multi-stream pipelines, the DUP System includes a simple coordination language which, similar to syntactic constructs in the UNIX shell, allows the user to specify how various stages should be connected with streams. The DUP runtime then sets up the streams and starts the various stages. Key benefits of the DUP System include:

1. Stages in a multi-stream pipeline can run in parallel and on different cores;
2. Stages can be implemented, compiled and tested individually using an appropriate language and compiler for the given problem and architecture;
3. Stages only communicate using streams; streams are a great match for networking applications and for modern processors doing sequential work;
4. If communication between stages is limited to streams, there is no possibility of data races and other issues that plague developers of parallel systems;
5. While the DUP System supports arbitrary data-flow graphs, the possibility of deadlocks can be eliminated by only using acyclic data-flow graphs;
6. Applications built using multi-stream pipelines can themselves be composed into a larger multi-stream pipeline, making it easy for programmers to express hierarchical parallelism

In addition to introducing the DUP System itself, this paper also presents experimental results from a case study involving the DUP System. The case study shows that it is possible to rapidly parallelize and distribute an existing complex legacy bioinformatics application and obtain significant speed-ups using DUP.

2 Approach

The fundamental goal of multi-stream pipelines is to allow processes to read from multiple input streams and write to multiple output streams, all of which may be connected to produce the desired data-flow graph. This generalization of linear

UNIX pipelines can be implemented using traditional UNIX APIs,[2] especially the `dup2` system call. Where a typical UNIX shell command invocation only connects `stdin`, `stdout` and `stderr`, the DUP System establishes additional I/O streams before starting a stage. Using this method, traditional UNIX filters (such as `grep`) can be used as stages in the DUP System without modification. New stages can be implemented in any language environment that supports POSIX-like input-output operations (specifically, reading and writing to a file). Since `dup2` also works with TCP sockets, the DUP System furthermore generalizes multi-stream pipelines to distributed multi-stream pipelines.

2.1 The DUP Assembly Language

The DUP Assembly language allows developers to specify precisely how to connect stages and where those stages should be run. Figure 1 lists the DUP Assembly code for a distributed "Hello World" example program.

```
s @10.0.0.1[0<in.txt,1|g1:0,3|g2:0]$ fanout;
g1@10.0.0.1[1|in:0]                $ grep Hello;
g2@10.0.0.2[1|in:3]                $ grep World;
in@10.0.0.2[1>out.txt]             $ faninany;
```

Fig. 1. DUP specification. `in.txt` is passed to `fanout` ("`0<in.txt`") which copies the stream to all outputs; in this case output 1 to stream 0 (≡ `stdin`) at `g1` ("`1|g1:0`") and output 3 to stream 0 at `g2` ("`3|g2:0`"). `g1` and `g2` run `grep`, the outputs (1 ≡ `stdout`) flowing into stage `in` as streams 0 and 3 respectively. `in` merges those streams and writes the output into `out.txt`. The resulting data flow is illustrated in Figure 3.

In essence, the DUP language allows developers to specify a directed graph using an adjacency list representation and IO redirection syntax similar to that of well-known UNIX shells [3]. The nodes in the directed graph are the stages initiated by DUP. A DUP program consists of a list of statements, each of which corresponds to one such node. Statements start with a label that is used to reference the respective stage in the specification of other stages. The keyword `DUP` is used to reference streams associated with the controlling `dup` command in the case that the `dup` command itself is used as a stage.

The label is followed by a hostname specifying on which system the stage will be run. A helper process, `dupd`, will be started on the specified host, listen on a port to establish network connections and eventually supervise stages run there. The address is followed by a comma-separated list of edges representing primarily the outgoing streams for this stage. Input streams are only explicitly specified in the case of input from files or the controlling `dup` command. Inputs from other stages are not specified because they can be inferred from the respective entry

[2] The APIs needed are supported by all platforms conforming to the POSIX standard, including BSD, GNU/Linux, OS X, and z/OS.

```
<PROGRAM>  ::= <STAGE>*
<STAGE>    ::= <LABEL> '@' <ADDRESS> '[' <EDGELIST> ']' '$' <COMMAND> ';'
<EDGELIST> ::= <EDGE> (',' <EDGE>)*
<EDGE>     ::= <INTEGER> <OP> <NODE>
<NODE>     ::= <REMOTEPR> | <UNIX_PATH>
<REMOTEPR> ::= <LABEL> ':' <INTEGER>
<OP>       ::= '|' | '<' | '>' | '>>'
```

Fig. 2. Simplified grammar for the DUP Assembly language. Note that we do not expect programmers to need to develop applications by *directly* using this language in the future; this language is the "assembly" language supported by the DUP runtime system. Higher-level languages running on top of DUP that facilitate (static) process scheduling and aspect oriented programming are under development.

of the producing stage. DUP supports four different ways to create streams for a stage:

Read An input file edge consists of an integer, the "<" operator and a path to the file to be used as input. The integer is the file descriptor from which the stage will read the input stream. `dupd` is responsible for opening the input stream and validating that the file exists and is readable.

Write An output file edge for writing consists of an integer, the ">" operator and a path to the file to be overwritten or created. The integer is the file descriptor to which this stage will write. `dupd` checks that the specified path can be used for writing.

Append An output file edge for appending consists of an integer, the ">>" operator and a path to the file. The integer is the file descriptor to which this stage will write.

Pipe Non-file output edges consist of an integer, the "|" operator, a stage label, the ":" character and another integer. The first integer specifies the file descriptor to which this stage will write. The label specifies the process on the other end of the pipe or TCP stream and the second integer is the file descriptor from which the other stage will read. If an edge list contains a label that is not defined elsewhere in the configuration file then the program file is considered malformed and rejected by `dup`.

The final component of a complete stage statement is the command (with arguments) that is used to start the process. Figure 2 contains a formal grammar for the DUP language. The grammar omits I/O redirection from/to the controlling `dup` command for clarity.

2.2 DUP System Architecture

The `dup` client interprets the mini-language from Section 2.1 which specifies how the various stages for the application should be connected. `dup` then connects to hosts running `ssh` servers and starts `dupd` helper processes which then receive control information via the SSH tunnel. The control information specifies the

binary names and arguments for the stages as well as how to establish TCP streams and UNIX pipes to connect the stages with each other.

Figure 3 illustrates how the components of the system work together.

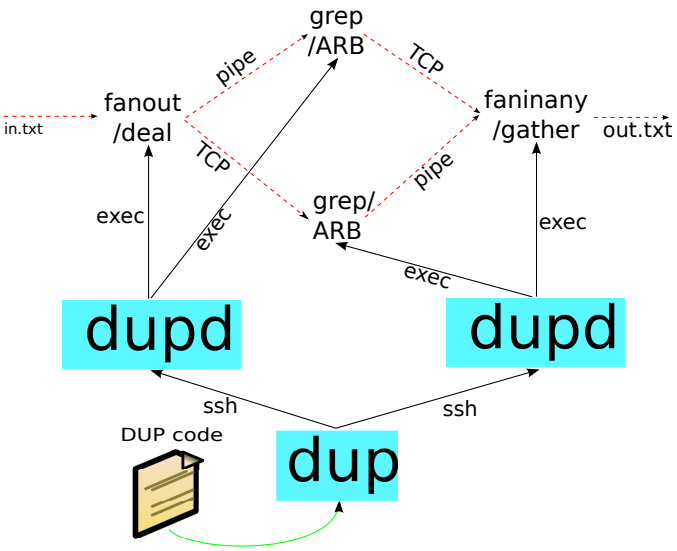

Fig. 3. Overview for one possible configuration of the DUP System. Red (dashed) lines show application data flow. Black (solid) lines correspond to actions by DUP. Examples for DUP Assembly corresponding to the illustration are given in Figures 1, 4 and 5 respectively: the three code snippets specify the same data-flow graph, but with different commands.

The primary interaction between `dup` and the `dupd`s involves four key steps [4]:

1. `dup` starts the `dupd`s and transmits session information. This includes all of the information related to processes that are supposed to be run on the respective `dupd`.
2. When a stage is configured to transmit messages to a stage initiated by another `dupd`, the `dupd` responsible for the data-producing stage establishes a TCP connection to the other `dupd` and transmits a header specifying which stage and file descriptor it will connect to the stream. If `dup` is used as a filter, it too opens similar additional TCP streams with the respective `dupd`s. The main difference here is that `dup` also initiates TCP connections for streams where `dup` will ultimately end up receiving data from a stage.
3. Once a `dupd` has confirmed that all required TCP streams have been established, that all required files could be opened, and that the binaries for the stages exist and are executable, it transmits a "*ready*" message to the controlling `dup` process (using the connection on which the session information was initially received).
4. Once all `dupd`s are ready, `dup` sends a "*go*" message to all `dupd`s. The `dupd`s then start the processes for the session.

2.3 Generic DUP Stages

Taking inspiration from stages available in CMS [5,6], the DUP System includes a set of fundamental multi-stream stages. UNIX already provides a large number of filters that can be used to quickly write non-trivial applications with a linear pipeline. Examples of traditional UNIX filters include `grep` [7], `awk` [8], `sed` [8], `tr`, `cat`, `wc`, `gzip`, `tee`, `head`, `tail`, `uniq`, `buffer` and many more [3].

While these standard tools can all be used in the DUP System, none of them support multiple input or output streams. In order to facilitate the development of multi-stream applications with DUP, we provide a set of primitive stages for processing multiple streams. Some of the stages currently included with the DUP System are summarized in Table 1. Many of the stages listed in Table 1 are inspired by the CMS multi-stream pipeline implementation [6]. Naturally, we expect application developers to write additional application-specific stages.

Table 1. Summary of general-purpose multi-stream stages to be used with DUP in addition to traditional UNIX filters. Most of the filters above can either operate line-by-line in the style of UNIX filters or using a user-specified record length.

Stage	Description	I/O Streams	
		in	out
`fanout`	Replicate input n times	1	n
`faninany`	Merge inputs, any order	n	1
`gather`	Merge inputs, round-robin (waits for input)	n	1
`holmerge`	Forward input from stream that has sent the most data so far, discard data from other streams until they catch up	n	1
`deal`	Split input round robin to output(s), or per control stream	2	n
`mgrep`	Like `grep`, except non-matching lines output to secondary stream	1	2
`lookup`	Read keys from stream 3; tokens to match keys from stream 0; write matched tokens to 1, unmatched to 4 and unmatched keys to 5	2	3
`gate`	forward 1st input to 1st output until 2nd input ready	2	1

2.4 DUP Programming Philosophy

In order to avoid the common data consistency issues often found in parallel programming systems, stages and filters for DUP should not perform any updates to shared storage outside of the memory of the individual process. While the DUP System has no way to enforce this property, updates to files or databases could easily cause problems; if stages were allowed to update storage, changes in the order of execution could easily result in unexpected non-determinism. This might be particularly problematic when network latency and stage scheduling cause non-deterministic runs in a larger system that replicates parts of the computation (e.g., in order to improve fault-tolerance).

For applications that require parallel access to shared mutable state, the DUP System can still be used to parallelize (and possibly distribute) those parts that lend themselves naturally to stream processing. Other parts of the code should then be designed to communicate with the DUP parts of the application through streams.

We specifically expect stages developed for the DUP System to be written in many different languages. This will be necessary so that the application can take advantage of the specialized resources available in heterogeneous multi-core or HPC systems. Existing models for application development on these systems often force the programmer to use a particular language (or small set of languages) for the entire application. For example, in a recent study of optimization techniques for CUDA code [9], twelve benchmark programs were modified by porting critical sections to the CUDA model. On average, these programs were only 14% CUDA-specific, yet the presence of CUDA sections limits the choice of languages and compilers for the entire program. The implications are clear: the use of a monolithic software architecture for programs designed to operate efficiently on high-performance hardware will severely restrict choices of development teams and possibly prevent them from selecting the most appropriate programming language and tool-chain for each part of a computation. Using the DUP System, developers will be able to compose larger applications from stages written in the most appropriate language available.

Another important use-case for DUP is the parallel and distributed execution of legacy code. In contrast to other new languages for parallel programming, which all too often advocate for large-scale (and often manual) program translation efforts, the DUP philosophy calls for writing thin wrappers around legacy code to obtain a streaming API. As we experienced in our case study, it is typically easy to adapt legacy applications to consume inputs from streams and to produce outputs as streams.

3 Case Study: Distributed Molecular Sequence String Matching

Exact and inexact string searching in gene sequence databases plays a central role in molecular biology and bioinformatics. Many applications require string searches, such as searching for gene sequence relatives and mining for PCR-primers or DNA-probes in DNA sequences [10,11,12]; both of these applications are important in the process of developing molecular diagnostic assays for pathogenic bacteria or viruses based upon specific DNA amplification and detection.

In the ARB software package, a suffix-tree-based search index, called the PT-Server, is the central data structure used by applications for fast sequence string matching [13]. A PT-Server instance is built once from the sequence entries of a gene sequence database of interest and is stored permanently on disk.

In order to perform efficient searches, the PT-Server is loaded into main memory in its entirety — if the entire data structure cannot fit into the available

main memory (the PT-Server requires $\sim$ 36 bytes of memory per sequence base), the database cannot be efficiently searched.

In addition to memory consumption, the runtime performance of the search can be quite computationally intensive. An individual *exact* string search — in practice, short sequence strings of length 15–25 base pairs are searched for — is quick (3–15 milliseconds). However, the execution time can become significant when millions of *approximate* searches are performed during certain bioinformatic analyses, such as probe design.

In the near future, the number of published DNA sequences will explode due to the availability of new high-throughput sequencing technology [14]. As a result, current sequential analysis methods will be unable to process the available data within reasonable amounts of time. Furthermore, rewriting more than half-a-million lines of legacy C and C++ code of the high-performance ARB software package is prohibitively expensive. The goal of this case study was to see how readily the existing ARB PT-Server could be distributed and parallelized using the DUP System. Specifically, we were interested in parallelization in order to reduce execution time and in distribution in order to reduce per-system memory consumption.

3.1 Material and Methods

The study used 16 compute nodes of the Infiniband Cluster in the Faculty of Informatics at the Technische Universität München [15]. Each node was equipped with an AMD Opteron 850 2.4 GHz processor with 8 GB of memory, and the nodes were connected using a 4x Infiniband network. The SILVA database (SSURef_91_SILVA_18_07_07_opt.arb) [16], which stores sequences of small subunit ribosomal ribonucleic acids and consists of 196,890 sequence entries (with 289,563,473 bases), was used for preparing test database sets and respective PT-Servers. We divided the original database into 1, 2, 4, 8, and 16 partitions, and a random sampling algorithm was used for composing the partitioned database sets (within each database analysis set, each partition is about the same size). The PT-Servers used in this study were created from these partitions. Table 2 characterizes the resulting partitions and PT-Servers.

For the queries, we selected 800 inverse sequence strings of rRNA-targeted oligonucleotide probe sequences of length 15–20 from probeBase, a database of published probe sequences [17]. Each retrieved sequence string has matches in the SILVA database and the respective PT-Server instance. Applying these real world query sequence strings ensured that every search request required non-trivial computation and communication. We generated four sets of inverse sequence strings (400 strings each) by random string distribution of the original dataset from probeBase, and every test run was performed with these four datasets. The presented performance values are the means of the four individually recorded runs.

Table 2. Resulting problem sizes for the different numbers of partitions. This table lists the average number of sequences and bases for the PT-Server within each partition and the resulting memory consumption for each PT-Server as well as the total memory consumption for all partitions.

# Part.	# Sequences	# MBases	Memory (MB)	
			Part.	total
1	196,890	289.6	1,430	1,430
2	98,445	144.7	745	1,489
4	49,222	72.4	402	1,609
8	24,611	36.2	231	1,849
16	12,305	18.1	145	2,327

3.2 Adapting ARB for DUP

In the ARB software package, `arb_probe` is a program which performs, per execution, one string search using the PT-Server when a search string and accompanying search parameters are specified (these are passed as command line arguments). For DUP, `arb_probe` had to be modified to read the parameters and the search string as a single line from `stdin` and pass one result set per line to `stdout`. It took one developer (who had experience with ARB but not DUP or distributed systems) about three hours to create the modified version `arb_probe_dup` and another two hours to compile DUP on the Infiniband Cluster, write adequate DUP scripts and perform the first run-time test. Debugging, testing, optimization and gathering of benchmark results for the entire case study was done in less than two weeks.

All searches were conducted using the program `arb_probe_dup` with similar parameters: `id 1 mcmpl 1 mmis 3 mseq ACGTACGT`. The first parameter (`id 1`) set the PT-Server ID; the second activated the reverse complement sequence (`mcmpl 1`). For each dataset and approach, the third parameter was used to perform an exact search (`mmis 0`) in order to find matches identical with the search string and an approximate search (`mmis 3`) in order to find all identical strings and all similar ones with maximum distance of three characters to the search string. The last parameter indicated the match sequence.

Figure 4 shows the DUP assembly code for the replicated run with two servers. Here, identical PT-Servers are used with the goal of optimizing execution time. Figure 5 shows the equivalent DUP assembly code for the partitioned setting. In this case, since each PT-Server only contains a subset of the overall database, all requests are broadcast to all PT-Servers using `fanout`.

3.3 Results and Discussion

As shown in Table 2, partitioning the original database into n partitions results in almost proportional reductions in per-node memory consumption: doubling the number of partitions means almost halving the memory consumption per

```
s @opt1[0<in.txt,1|r1:0,3|r2:0] $ deal;
r1@opt1[1|re:0]                 $ arb_probe_dup;
r2@opt2[1|re:3]                 $ arb_probe_dup;
re@opt2[1>out.txt]              $ faninany;
```

Fig. 4. DUP specification for the replicated configuration that uses identical ARB servers. The queries are simply distributed round-robin over the two available ARB PT-Servers and the results collected as they arrive.

```
s @opt1[0<in.txt,1|p1:0,3|p2:0] $ fanout;
p1@opt1[1|pe:0]                 $ arb_probe_dup;
p2@opt2[1|pe:3]                 $ arb_probe_dup;
pe@opt2[1>out.txt]              $ gather;
```

Fig. 5. DUP specification for the partitioned configuration where each ARB server only contains a slice of the database. The queries are broadcast to the available ARB PT-Servers and the results collected in round-robin order (to ensure that results for the same query arrive in one batch).

PT-Server partition. In practice we expect significantly larger databases to be partitioned, resulting in partition sizes close to the size of the main memory of the HPC node responsible for the partition.

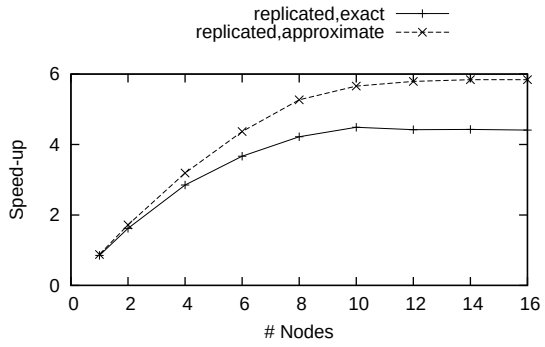

Fig. 6. Speedup of sequence string matches for the replicated PT-Server. The plot shows the average speed-up over five runs for an exact search and an approximate search (with up to three mismatches).

Figure 6 summarizes the speedup we obtained using n PT-Server replicas (each processing a fraction of the queries). This answers the question as to how much performance could be gained by distributing the queries over n identical

(replicated) PT-servers, each containing the full database. Compared with a local version (direct communication between a PT-Server and arb_probe_dup) we have measured a speedup of 5.84 for 16 compute nodes.

The available bandwidth of the compute cluster using TCP is about 107 MB/s, which is close to the 85 MB/s consumed on average by the collector node for 16 compute nodes. For this run, the average CPU utilization of the 16 compute nodes is about 34% and the master node uses about 56%. The legacy ARB code produces rather verbose output, which explains why this benchmark is IO-bound on our cluster when run with more than approximately 8 compute nodes. Converting the human-readable output to a compact binary format would likely result in a significant performance improvement; however, the purpose of this study was to evaluate possible speed-ups for legacy code without significant changes to the existing infrastructure and changing the message format of the ARB framework would be a significant change.

The overall runtime for querying a partitioned PT-Server with one sequence string set (400 requests) was in a range of 2 seconds (16 partitions) to 8.25 seconds (one partition) for exact searches, and 16 seconds (16 partitions) to 73 seconds (one partition) for approximate searches. For the replicated PT-Servers, execution time for exact searches ranged from approximately 8.3 seconds on one node to 1.5 seconds on 16 nodes. The approximate search (up to three mismatches) ranged from 72 seconds on one node to 13 seconds on 16 nodes. In an additional test run with the replicated servers, we increased the number of requests (to 2000) by repeating the string set to increase the measured time and reduce possible side effects. The execution time ranged from 140.91 seconds (one node) to 27.09 seconds (16 nodes) for exact searches, and 1479.60 seconds (one node) and 222.26 (16 nodes) for the approximate search.

3.4 Conclusion and Future Work

The speed-ups achieved in this case study are by themselves clearly not sensational; however, the ratio of speedup to development time is. Programmer productivity is key here, especially since researchers in bioinformatics are rarely also experts in distributed systems. Furthermore, the improvements in performance and memory consumption are significant and have direct practical value for molecular biologists and bioinformaticians, especially since, aside from the acceleration of sequence string searches by a factor 3.5 to 5.8, this approach also offers biologists the possibility to search very large databases using the ARB PT-Server without having to access special architectures with extreme extensions to main memory.

In the future, we plan to use DUP to drive large-scale bioinformatics analyses. Depending on the problem size, we also expect to use DUP to combine partitioning and replication in one system. For example, it would be easy to create n replicas of m partitions in order to improve throughput while also reducing the memory consumption of the PT-Servers. Finally, assuming additional performance is desired, the ARB data format could be changed to be less verbose and thereby avoid bandwidth limitations.

4 Related Work

The closest work to the DUP System presented in this paper are multi-stream pipelines in CMS [6]. CMS multi-stream pipelines provide a simple mini-language for the specification of virtually arbitrary data-flow graphs connecting stages from a large set of pre-defined tools or arbitrary user-supplied applications. The main difference between CMS and the DUP System (which uses parallel execution of stages) is that CMS pipelines are exclusively record-oriented and implemented through co-routines using deterministic and non-preemptive scheduling with zero-copy data transfer between stages. CMS pipelines were designed for efficient execution in a memory-constrained, single-tasking operating system with record-oriented files. In contrast, DUP is designed for modern applications that might not use record-oriented I/O and need to run in parallel and on many different platforms.

Another close relative to the DUP System are Kahn Process Networks (KPNs) [18]. A major difference between DUP and KPNs is that buffers between stages in DUP are bounded, which is necessary given that unbounded buffers cannot really be implemented and that in general determining a bound on the necessary size of buffers (called channels in KPN terminology) is undecidable [19]. Note that the UNIX command `buffer` can be used to create buffers of arbitrary size between stages in DUP. Another major difference with KPNs is that DUP does not require individual processes to be deterministic. Non-determinism on the process level voids some of the theoretical guarantees of KPNs; however, it also enables programmers to be much more flexible in their implementations. While DUP allows non-determinism, DUP programmers explicitly choose non-deterministic stages in specific places; as a result, non-determinism in DUP is less pervasive and easier to reason about compared to languages offering parallel execution with shared memory.

Where CMS pipelines focus on the ability to glue small, reusable programs into larger applications, the programming language community has extended various general-purpose languages and language systems with support for pipelines. Existing proposals for stream-processing languages have focused either on highly-efficient implementation (for example, for the data exchange between stages [20]) or on enhancing the abstractions given to programmers to specify the pipeline and other means of communication between stages [21]. The main drawback of all of these designs is that they force programmers to learn a complex programming language and rewrite existing code to fit the requirements of the particular language system. The need to follow a particular paradigm is particularly strong for real-time and reactive systems [22,23]. Furthermore, especially when targeting heterogeneous multi-core systems, quality implementations of the particular language must be provided for each architecture. In contrast, the DUP language implementation is highly portable (relying exclusively on canonical POSIX system calls) and allows developers to implement stages in any language.

On the systems side, related research has focused on maximizing performance of streaming applications. For example, StreamFlex [22] eliminates copying between filters and minimizes memory management overheads using types. Other

research has focused on how filters should be mapped to cores [24] or how to manage data queues between cores [20]. While the communication overheads of DUP applications can likely be improved, this could not be achieved without compromising on some of the major productivity features of the DUP System (such as language neutrality and platform independence).

In terms of language design and runtime, the closest language to the DUP Assembly language is Spade [25] which is used to write programs for InfoSphere Streams, IBM's distributed stream processing system [26]. The main differences between Spade and the DUP Assembly language is that Spade requires developers to specify the format of the data stream using types and has built-in computational operators. Spade also restricts developers of filters to C++; this is largely because the InfoSphere runtime supports migrating of stages between systems for load-balancing and can also fuse multiple stages for execution in a single address space for performance. Dryad [27] is another distributed stream processing system similar to Spade in that it also restricts developers to developing filters in C++. Dryad's scheduler and fault-tolerance provisions further require all filters to be deterministic and graphs to be free of cycles, making it impossible to write stages such as `faninany` or `holmerge` in Dryad. In comparison to both Spade and Dryad, the DUP System provides a simpler language with a much more lightweight and portable runtime system. DUP also does not require the programmer to specify a specific stream format, which enables the development of much more generic stages. Specifically, the Spade type system cannot be used to properly type stream-format agnostic filters such as `cat` or `fanout`. Finally, DUP is publicly available whereas both Spade and Dryad are proprietary.

DUP is a coordination language [28] following in the footsteps of Linda [29]: the DUP System is used to coordinate computational blocks described in other languages. The main difference between DUP and Linda is that in DUP, the developer specifies the data flow between the components explicitly, whereas in Linda, the Linda implementation needs to match tuples published in the tuplespace against tuples published by other components. The matching of tuples in the Linda system enables Linda to execute in a highly dynamic environment where processes joining and leaving the system are easily managed. However, the matching and distribution of tuples also causes significant performance issues for tuplespace implementations [30]. As a result, Linda implementations are not suitable for distributed stream processing with significant amounts of data.

5 Conclusion

The significant challenges with writing efficient parallel high-performance code are numerous and well-documented. The DUP System presented in this paper addresses some of these issues using multi-stream pipelines as a powerful and flexible abstraction around which an overall computation can be broken into independent stages, each developed in the language best suited for the stage, and each compiled or executed by the most effective tools available. Our experience so

far makes us confident that DUP can be used to quickly implement parallel programs, to obtain significant performance gains, and to experiment with various dataflow graph configurations with different load-distribution characteristics.

Acknowledgements

The authors thank Prof. Dr. Matthias Horn, University of Vienna, for providing us with probe sequences from probeBase for the bioinformatics case study. This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG) under ENP GR 3688/1-1 and by the Bayerische Forschungsstiftung (BFS) under AZ 767-07 (the NANOBAK Projekt).

References

1. Lindholm, E., Nickolls, J., Oberman, S., Montrym, J.: NVIDIA Tesla: A unified graphics and computing architecture. IEEE Micro **28** (2008) 39–55
2. Flachs, B., Asano, S., Dhong, S.H., Hofstee, P., Gervias, G., Kim, R., Le, T., Liu, P., Leenstra, J., Liberty, J., Michael, B., Oh, H., Mueller, S.M., Takahashi, O., Hatakeyama, A., Wantanbe, Y., Yano, N.: A stream processing unit for a cell processor. In: IEEE International Solid-State Circuits Conference. (2005) 134–135
3. Quigley, E.: UNIX Shells. 4 edn. Prentice Hall (2004)
4. Grothoff, C., Keene, J.: The DUP protocol specification v2.0. Technical report, The DUP Project (2010)
5. Hartmann, J.P.: CMS Pipelines Explained. IBM Denmark, `http://vm.marist.edu/~pipeline/`. (2007)
6. IBM: CMS Pipelines User's Guide. version 5 release 2 edn. IBM Corp., `http://publibz.boulder.ibm.com/epubs/pdf/hcsh1b10.pdf` (2005)
7. Goebelbecker, E.: Using grep: Moving from DOS? Discover the power of this Linux utility. Linux Journal (1995)
8. Dougherty, D.: Sed and AWK. O'Reilly & Associates, Inc., Sebastopol, CA, USA (1991)
9. Ryoo, S., Rodrigues, C.I., Baghsorkhi, S.S., Stone, S.S., Kirk, D.B., mei W. Hwu, W.: Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2008) 73–82
10. Nordberg, E.K.: YODA: selecting signature oligonucleotides. Bioinformatics **21** (2005) 1365–1370
11. Linhart, C., Shamir, R.: The degenerate primer design problem. Bioinformatics **18 Suppl 1** (2002) S172–181
12. Kaderali, L., Schliep, A.: Selecting signature oligonucleotides to identify organisms using DNA arrays. Bioinformatics **18** (2002) 1340–1349
13. Ludwig, W., Strunk, O., Westram, R., Richter, L., Meier, H., Yadhukumar, Buchner, A., Lai, T., Steppi, S., Jobb, G., Förster, W., Brettske, I., Gerber, S., Ginhart, A.W., Gross, O., Grumann, S., Hermann, S., Jost, R., König, A., Liss, T., Lüssmann, R., May, M., Nonhoff, B., Reichel, B., Strehlow, R., Stamatakis, A., Stuckmann, N., Vilbig, A., Lenke, M., Ludwig, T., Bode, A., Schleifer, K.H.: ARB: a software environment for sequence data. Nucleic Acids Research **32** (2004) 1363–1371

14. Shendure, J., Ji, H.: Next-generation DNA sequencing. Nat. Biotechnol. **26** (2008) 1135–1145
15. Klug, T.: (Hardware of the InfiniBand Cluster)
16. Pruesse, E., Quast, C., Knittel, K., Fuchs, B.M., Ludwig, W., Peplies, J., Glöckner, F.O.: SILVA: a comprehensive online resource for quality checked and aligned ribosomal RNA sequence data compatible with ARB. Nucleic Acids Research **35** (2007) 7188–7196
17. Loy, A., Maixner, F., Wagner, M., Horn, M.: probeBase – an online resource for rRNA-targeted oligonucleotide probes: new features 2007. Nucleic Acids Research **35** (2007)
18. Kahn, G.: The semantics of a simple language for parallel programming. Information Processing (1974) 993–998
19. Parks, T.M.: Bounded Scheduling of Process Networks. PhD thesis, University of California, Berkeley (1995)
20. Giacomoni, J., Moseley, T., Vachharajani, M.: Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2008) 43–52
21. Thies, W., Karczmarek, M., Amarasinghe, S.P.: Streamit: A language for streaming applications. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, London, UK, Springer-Verlag (2002) 179–196
22. Spring, J.H., Privat, J., Guerraoui, R., Vitek, J.: Streamflex: high-throughput stream programming in java. SIGPLAN Not. **42** (2007) 211–228
23. Lee, E.A.: Ptolemy project. `http://ptolemy.eecs.berkeley.edu/` (2008)
24. Kudlur, M., Mahlke, S.: Orchestrating the execution of stream programs on multicore platforms. In: PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2008) 114–124
25. Hirzel, M., Andrade, H., Gedik, B., Kumar, V., Losa, G., Soule, R., Kun-Lung-Wu: Spade language specification. Technical report, IBM Research (2009)
26. Amini, L., Andrade, H., Bhagwan, R., Eskesen, F., King, R., Selo, P., Park, Y., Venkatramani, C.: Spc: A distributed, scalable platform for data mining. In: Workshop on Data Mining Standards, Services and Platforms (DM-SPP). (2006)
27. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. In: European Conference on Computer Systems (EuroSys), Lisabon, Portugal (2007) 59–72
28. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM **35** (1992) 97–107
29. Carriero, N., Gelernter, D.: Linda in context. Commun. ACM **32** (1989) 444–458
30. Wells, G.C.: A Programmable Matching Engine for Application Development in Linda. PhD thesis, University of Bristol (2001)