Toward Extreme-Scale High-Performance Computing Using a Fine-Grain Dataflow-Inspired Execution Model

Stéphane Zuckerman

Computer Architecture & Parallel Systems Laboratory
Electrical & Computer Engineering Dept.
University of Delaware
140 Evans Hall Newark,DE 19716, United States

September 10, 2014

Outline

Outline

1. **The State of the High-Performance Computing World in 2005–2010**
2. **A Short Introduction to Execution Models**
   - The Von Neumann Model
   - The Dataflow Model
3. **The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems**
4. **DARTS: An Implementation of the Codelet Model**
   - DARTS: Implementation of the Codelet Machine Model
   - DARTS: Experimental Results
     - Running DGEMM in DARTS
     - Running Graph500 in DARTS
5. **Running DARTS on a Dataflow-Enabled Multi-Core Architecture**
   - The TERAFLUX Project
   - Porting DARTS to COTSon
   - DARTS/COTSon: Experimental Results
6. **The Future of Codelets**

2004–2005: Apparition of Multi-Core Systems

- The power wall leads to the first multi-core processors
- Memory wall: a major performance issue (See Wulf and McKee 1995)
- GPUs become more programmable (but still through dirty hacks)

2004–2005: Apparition of Multi-Core Systems

- The power wall leads to the first multi-core processors
- Memory wall: a major performance issue (See Wulf and McKee 1995)
- GPUs become more programmable (but still through dirty hacks)

2006–2007: Real Multi-Core Processors Appear

- Intel proposes "real" multi-core processors (but still use a front-side bus)
- AMD provides an efficient interconnect for NUMA architectures
- IBM unveils the POWER6, Cell B.E. and Cyclops-64
- Nvidia uncovers CUDA (No need to resort to dirty hacks anymore)

2004–2005: Apparition of Multi-Core Systems

- The power wall leads to the first multi-core processors
- Memory wall: a major performance issue (See Wulf and McKee 1995)
- GPUs become more programmable (but still through dirty hacks)

2006–2007: Real Multi-Core Processors Appear

- Intel proposes “real” multi-core processors (but still use a front-side bus)
- AMD provides an efficient interconnect for NUMA architectures
- IBM unveils the POWER6, Cell B.E. and Cyclops-64
- Nvidia uncovers CUDA (No need to resort to dirty hacks anymore)

2008–2010: Toward “Many-Core” Compute Nodes

- Compute nodes start to propose a large number of cores
  - *e.g.*, 8-Core Intel Nehalem EX: 4×16 threads per node, with a NUMA Interconnect
- Nvidia commercializes boards dedicated to supercomputing

Meanwhile, in Versailles. . .

- 2006: Compiler transformation – Deep Jam
- 2007–2008: Methodology to fine-tune kernels on multicore systems
- 2009–2010: A balanced approach to application performance tuning
- 2010: Tackling cache line stealing in multicore systems

See Carribault et al. 2007; Zuckerman, Pérache, and Jalby 2008; Koliaï et al. 2009; Risio et al. 2009; Charif Rubial et al. 2009; Zuckerman and Jalby 2010

Meanwhile, in Versailles...

- 2006: Compiler transformation – Deep Jam
- 2007–2008: Methodology to fine-tune kernels on multicore systems
- 2009–2010: A balanced approach to application performance tuning
- 2010: Tackling cache line stealing in multicore systems

See Carribault et al. 2007; Zuckerman, Pérache, and Jalby 2008; Koliaï et al. 2009; Risio et al. 2009; Charif Rubial et al. 2009; Zuckerman and Jalby 2010

Main Parallel Programming Models

- MPI
- OpenMP
- CUDA

...for adventurers only

What to Expect for the Next Generation HPC Systems?

- Core/thread count per processor is rising
- Amount of cache per core/thread is decreasing
- Memory is becoming a *severe* bottleneck
  - Many people think coherence will have to go

Parallel Programming in 2005–2010

Meanwhile, in Versailles. . .

- 2006: Compiler transformation – Deep Jam
- 2007–2008: Methodology to fine-tune kernels on multicore systems
- 2009–2010: A balanced approach to application performance tuning
- 2010: Tackling cache line stealing in multicore systems

See Carribault et al. 2007; Zuckerman, Pérache, and Jalby 2008; Koliaï et al. 2009; Risio et al. 2009; Charif Rubial et al. 2009; Zuckerman and Jalby 2010

Main Parallel Programming Models

- MPI
- OpenMP
- CUDA

. . . for adventurers only

What to Expect for the Next Generation HPC Systems?

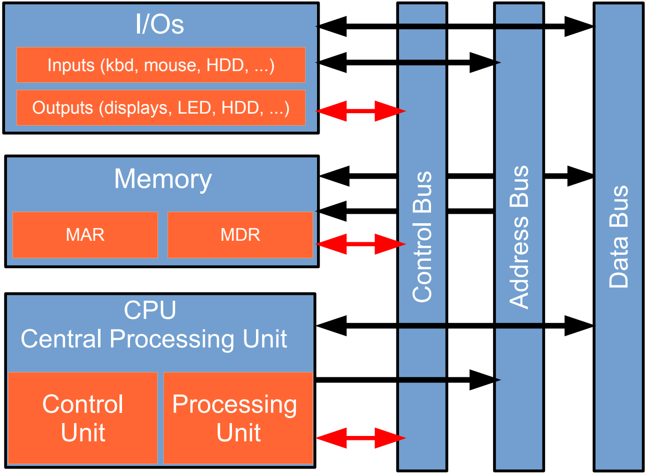
- Core/thread count per processor is rising
- Amount of cache per core/thread is decreasing
- Memory is becoming a *severe* bottleneck
  - Many people think coherence will have to go

How will we program the next parallel processors?

Outline

A Short Introduction to Execution Models

The von Neumann Model – a High-Level View



Advantages of the von Neumann Model

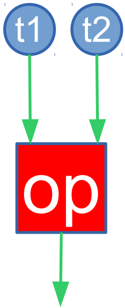
- Simple
- Can almost be implemented "directly"
  - However nobody would design a processor this way nowadays

Limitations of the von Neumann Model

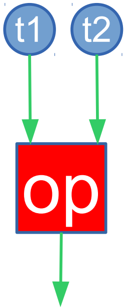
- Relies on a *sequence* of instructions
- Time is thus an integral part of the model
- Makes use of an accumulator: side-effects are inherent to the model
  - Reduces the potential for parallelism

Working Around Those Limitations

- Duplicate several "von Neumann machines," each with their own PC
- Add buses to share both memory and I/Os between processors
- . . . Is it still a von Neumann machine then?



t1
t2
op

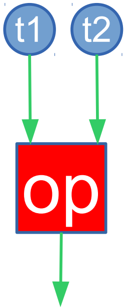


Static Dataflow Actors

Components of a regular actor:

- Input arcs which may contain at most 1 token each
- Output arcs which may contain at most 1 token each
- The operation provided by the actor
- Tokens

See Dennis, Fosseen, and Linderman 1972; Dennis 1974; Dennis and Misunas 1974



Static Dataflow Actors

Components of a regular actor:

- Input arcs which may contain at most 1 token each
- Output arcs which may contain at most 1 token each
- The operation provided by the actor
- Tokens

See Dennis, Fosseen, and Linderman 1972; Dennis 1974; Dennis and Misunas 1974

Firing Rule: Static Dataflow

An actor may *fire* when:

- All of its input arcs contain a token, and
- Its output arcs are empty.

Static Dataflow Actors

Components of a regular actor:

- Input arcs which may contain at most 1 token each
- Output arcs which may contain at most 1 token each
- The operation provided by the actor
- Tokens

See Dennis, Fosseen, and Linderman 1972; Dennis 1974; Dennis and Misunas 1974

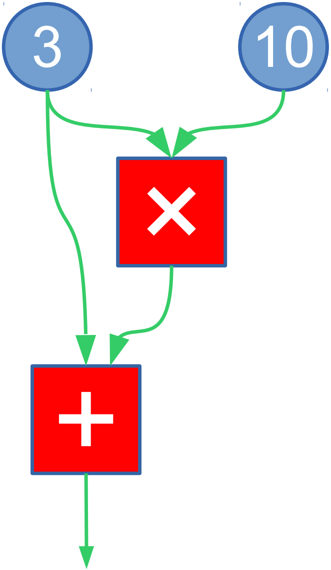
Firing Rule: Static Dataflow

An actor may *fire* when:

- All of its input arcs contain a token, and
- Its output arcs are empty.

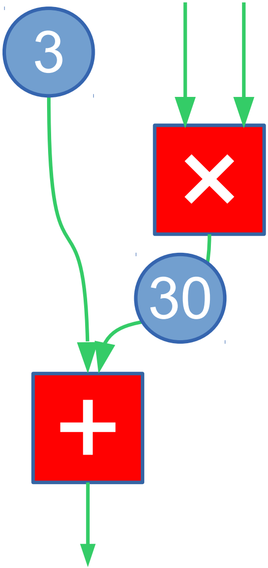
A Short Introduction to Execution Models

An Example of Dataflow Program



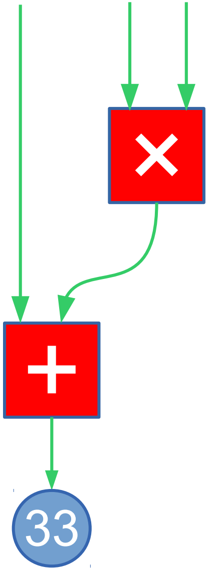
A Short Introduction to Execution Models

An Example of Dataflow Program



A Short Introduction to Execution Models

An Example of Dataflow Program



A Short Introduction to Execution Models

The Architecture Model of Static Dataflow

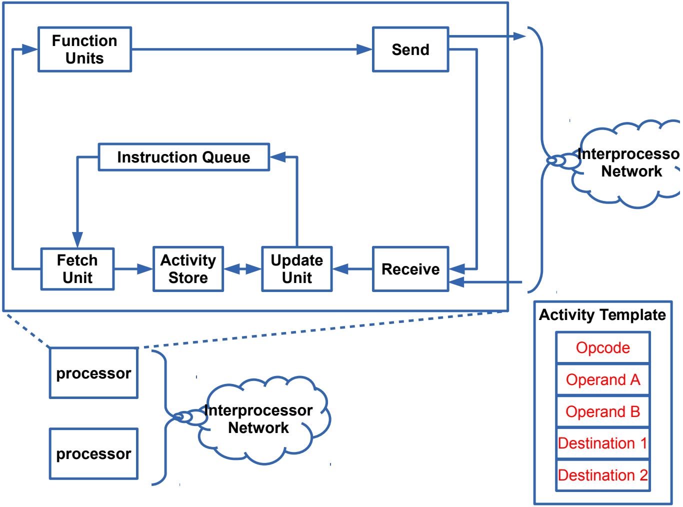
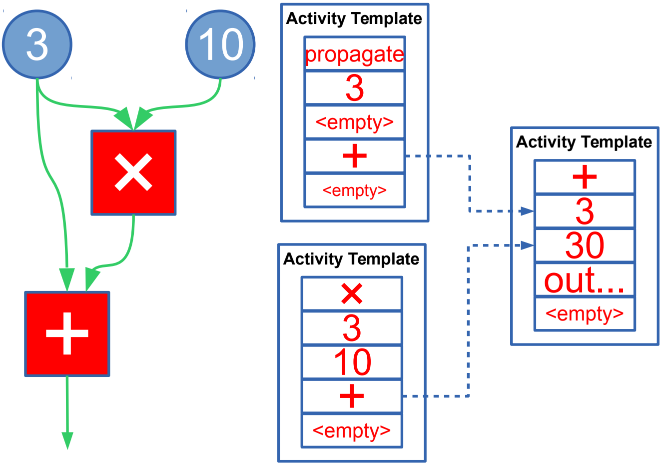


Figure : Inspired by J.Dennis' article (Encyclopedia of Parallel Computing)

Applying Our Example on the Static-DF Arch.



Outline

The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems

Objectives

- Fine-grain parallelism
- Scalable
- Expose maximal parallelism
- Limits non-determinism (determinate-by-default)
- Handles dynamic events (power, resiliency, resource constraints in general)

Definition

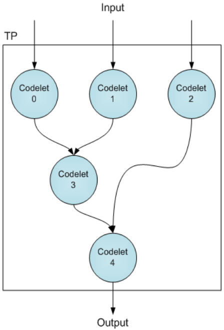
A codelet is a sequence of machine instructions which act as an atomically-scheduled unit of computation.

See DARPA-BAA-10-37 2010-2012; Carter et al. 2013; Department of Energy 2012–2014

The Codelet Model: Harnessing Parallelism in Shared-Memory Multi/Many Core Systems

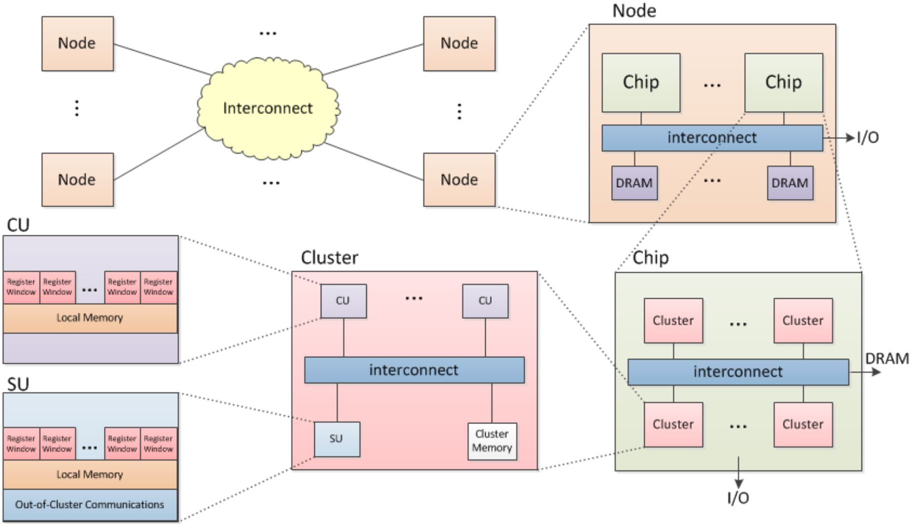
Properties

- Event-driven (availability of data and resources)
- Communicates only through its inputs and outputs
- Non-preemptive (with very *specific* exceptions)
- Requires all data and code to be “local”



See Zuckerman, Suetterlein, et al. 2011

The Codelet Abstract Machine



See Zuckerman, Suetterlein, et al. 2011

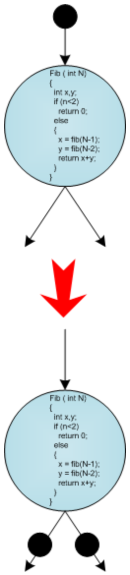
Codelet Graphs: Operational Semantics

Codelet Firing Rule

- Codelet actors are *enabled* once tokens are on each input arc
- Codelet actors fire by
  - consuming tokens
  - performing the operations within the codelet
  - producing a token on each of its output arcs

States of a Codelet

- Dormant: Not all tokens are available
- Enabled: All *data* tokens are available
- Ready: All tokens are available
- Active: The codelet is executing internal operations

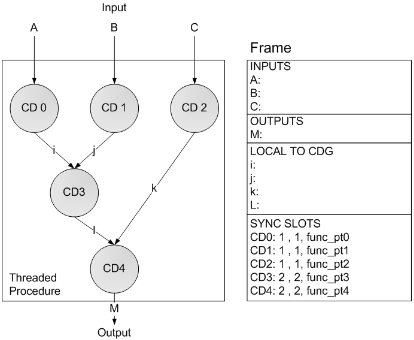


See Zuckerman, Suetterlein, et al. 2011

TPs are containers for codelet graphs, with additional meta-data.

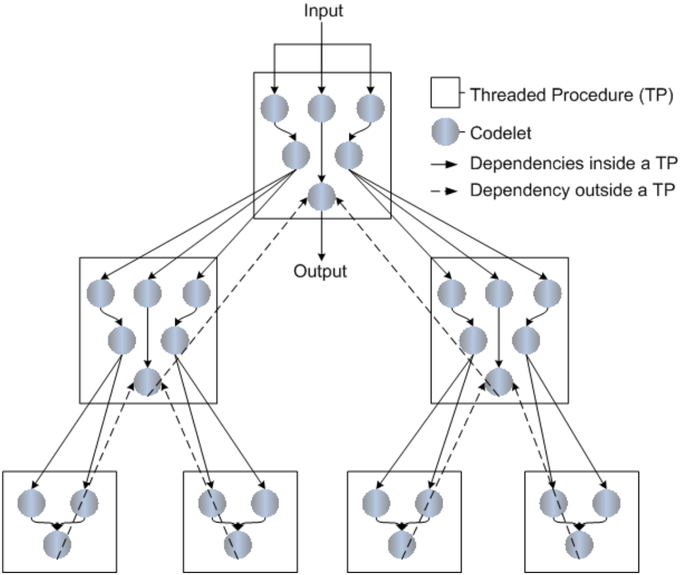
Description

- Invoked in a control-flow manner
- Called by a codelet from another CDG
- Feature a frame which contains the context of the CDG



See Zuckerman, Suetterlein, et al. 2011

An Example of Computation Using Threaded Procedures



See Zuckerman, Suetterlein, et al. 2011

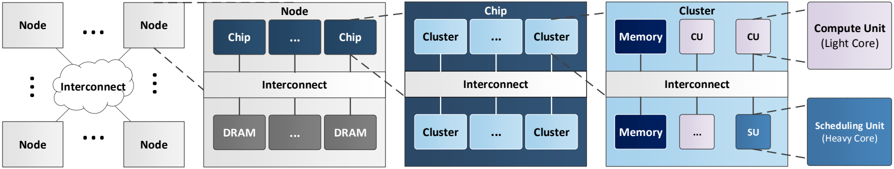
Objectives

- Faithfulness to the codelet execution model
- Modularity
  - So that portions of the runtime can be added or changed easily
  - For example: we have several codelet schedulers from which to choose
- Portability: Object-oriented, written in C++98, and makes use of open-source libraries:
  - `hwloc`: to determine the topology of the underlying system (HW threads/cores, caches, *etc.*)
  - If present on the system, it uses Intel TBB's lock-free queues

See Suetterlein, Zuckerman, and Gao 2013

DARTS: Implementation of the Codelet Machine Model

- Computation Units (CUs) embed a single producer/consumer ring buffer to store ready codelets
- Synchronization Units (SUs) embed two pools: Threaded Procedures and ready codelets.
- Heavy reliance on lock-free data structures
- SUs can temporarily assume the role of CUs if all other CUs are busy and there are ready codelets left to execute.



See Suetterlein, Zuckerman, and Gao 2013

Experimental Setup

AMD Opteron 6234 (Bulldozer) – Mills – 128 GiB DDR DRAM

		Cache Level	Shared By	Size (KiB)
Clock (GHz)	2.4	L1 Data	1 core	16
Threads / core	1	L1 Instruction	1 core	64
Cores / socket	12	L2 Unified	2 cores	2048
Sockets / node	4	L3 Unified	6 cores	6144
Compiler		gcc v4.6		
Math Library		AMD Core Math Library (ACML) v5.3		

Note: FPUs are shared between 2 cores.

Intel Xeon E5-2670 (Sandy Bridge) – FatNode – 64 GiB DDR3 DRAM

		Cache Level	Shared By	Size (KiB)
Clock (GHz)	2.6	L1 Data	2 threads	32
Threads / core	2	L1 Instruction	2 threads	32
Cores / socket	8	L2 Unified	2 threads	256
Sockets / node	2	L3 Unified	8 threads	20480
Compiler		gcc v4.7		
Math Library		Intel Math Kernel Library (MKL) v11.1		

Note: Functional units are shared between 2 threads.

Running DGEMM in DARTS – Codelet Graph

Description of DGEMM

- **D**ouble precision **GE**neral **M**atrix **M**ultiplication
- Used ACML or MKL as sequential building blocks (no tiling/blocking, *etc.*, needed)
- We compared several codelet scheduling policies within a cluster of cores

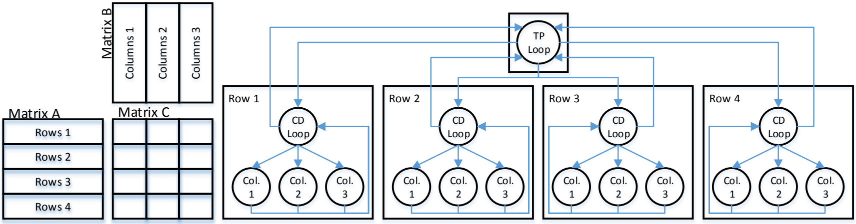


Figure : Our Codelet Graph decomposition for a parallel DGEMM

See Suetterlein, Zuckerman, and Gao 2013

Running DGEMM in DARTS – Mills – Strong Scaling

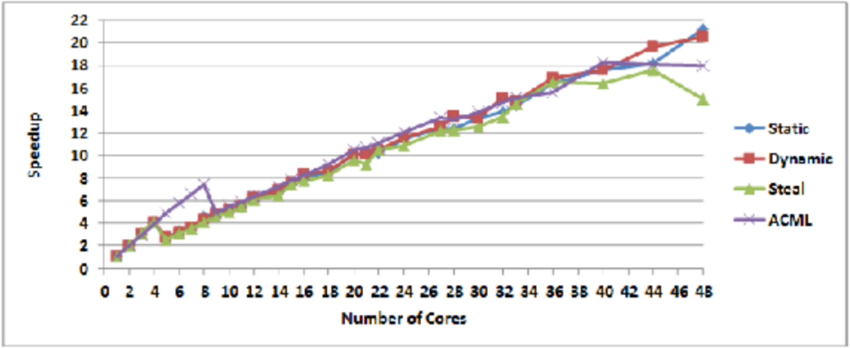


Figure : 10000×10000 Square DGEMM – Strong Scaling.

See Suetterlein, Zuckerman, and Gao 2013

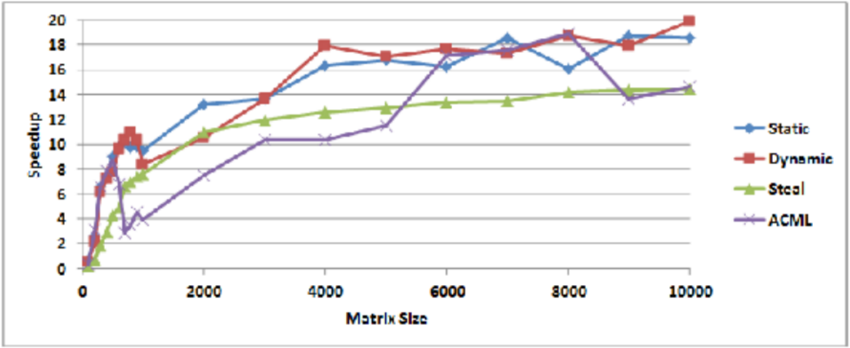


Figure : 48 cores – Square DGEMM – Weak Scaling.

See Suetterlein, Zuckerman, and Gao 2013

Running DGEMM in DARTS – FatNode – Strong Scaling

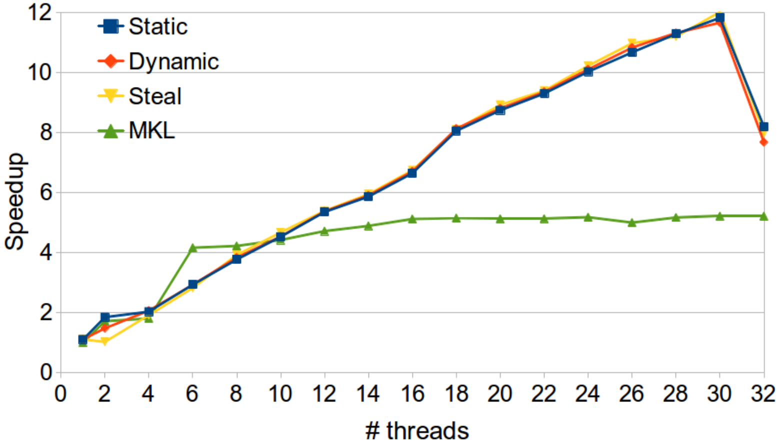


Figure : 3072×3072 Square DGEMM – Strong Scaling.

Running DGEMM in DARTS – FatNode – Weak Scaling

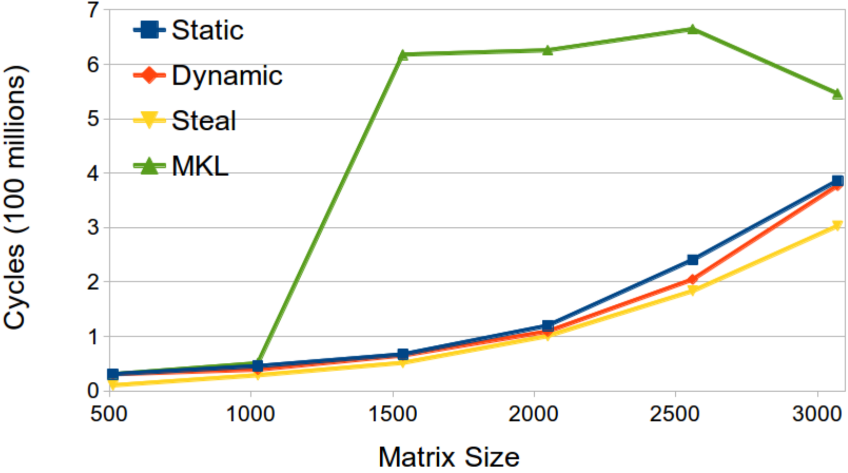
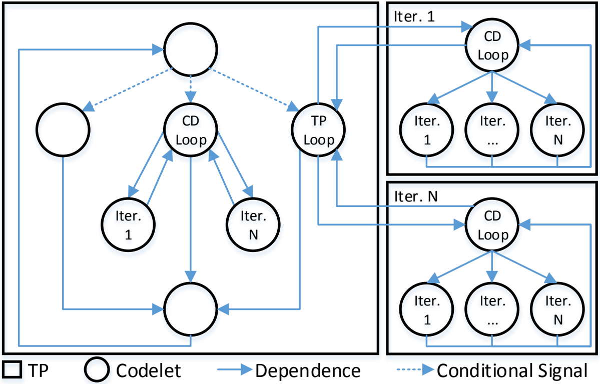


Figure : 32 threads – Square DGEMM – Weak Scaling.

Running Graph500 in DARTS – Codelet Graph

Description of Graph500

- Reused reference code (`http://graph500.org`)
- Only modified the breadth-first search phase (BFS)
- Compared with reference OpenMP parallelization
- Unit: Traversed Edges Per Second (TEPS)



See Suetterlein, Zuckerman, and Gao 2013

Running Graph500 in DARTS – Mills – Strong Scaling

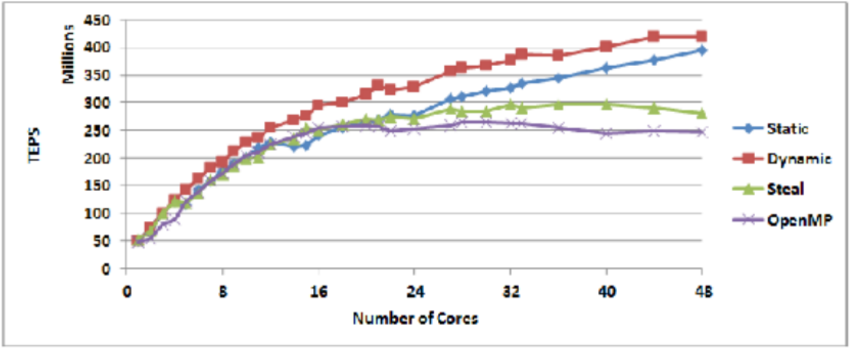


Figure : $Scale = 2^{18}$ – Graph500 – Strong Scaling

See Suetterlein, Zuckerman, and Gao 2013

Running Graph500 in DARTS – Mills – Weak Scaling

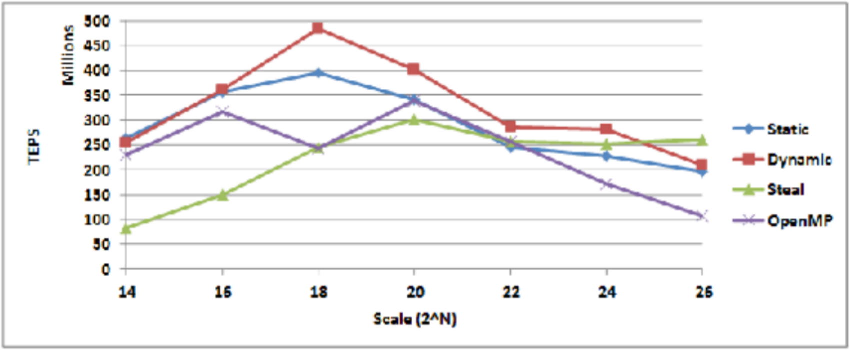


Figure : 48 cores – Graph500 – Weak Scaling

See Suetterlein, Zuckerman, and Gao 2013

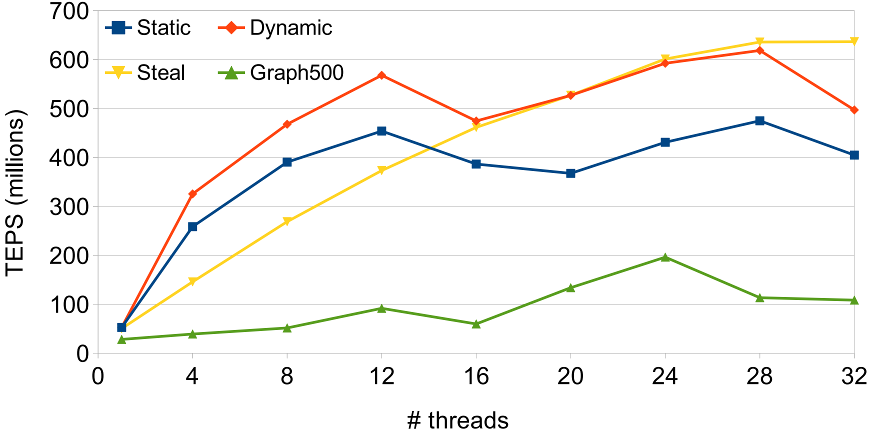


Figure : $Scale = 2^{18}$ – Graph500 – Strong Scaling

See Suetterlein, Zuckerman, and Gao 2013

Running Graph500 in DARTS – FatNode – Weak Scaling

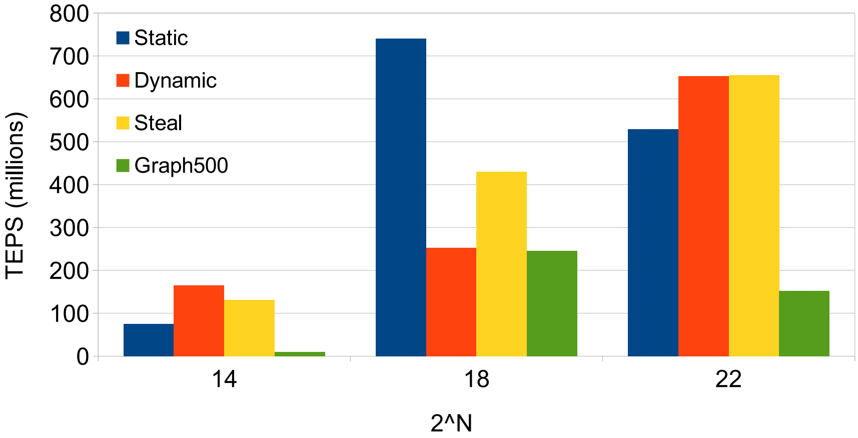


Figure : 32 cores – Graph500 – Weak Scaling

See Suetterlein, Zuckerman, and Gao 2013

Outline

Project Objectives

Future Teradevice systems will expose a large amount of parallelism (1000+ cores) that cannot be exploited efficiently by current applications and programming models. The aim of this project is to propose a complete solution that is able to harness the large-scale parallelism in an efficient way. The main objectives of the project are the programming model, compiler analysis, and a scalable, reliable, architecture based mostly on commodity components. Data-flow principles are exploited at all levels as to overcome the current limitations.

For more details, see `http://teraflux.eu`

See Solinas et al. 2013; Giorgi et al. 2014; Zuckerman, Arteaga, et al. 2014

The DF-Thread Model

- A *DataFlow Thread* (DF-Thread) is a non-preemptive piece of code which is ready to be *fired* when all its data dependencies are met.
- A DF-Frame contains all the data required by the DF-Thread to run.
  - While there are dependencies left, a DF-Frame is write-only
  - Once all dependencies are met, the frame becomes read-only
- The TERAFLUX abstract machine model features:
  - A Thread Scheduling Unit (equivalent of the Codelet Model's SU)
  - A Fault-Detection Unit (to handle fault-tolerance)

See Solinas et al. 2013; Giorgi et al. 2014; Zuckerman, Arteaga, et al. 2014

The DF-Thread Model

- A *DataFlow Thread* (DF-Thread) is a non-preemptive piece of code which is ready to be *fired* when all its data dependencies are met.
  - I've heard that line somewhere...
- A DF-Frame contains all the data required by the DF-Thread to run.
  - While there are dependencies left, a DF-Frame is write-only
  - Once all dependencies are met, the frame becomes read-only
- The TERAFLUX abstract machine model features:
  - A Thread Scheduling Unit (equivalent of the Codelet Model's SU)
  - A Fault-Detection Unit (to handle fault-tolerance)

See Solinas et al. 2013; Giorgi et al. 2014; Zuckerman, Arteaga, et al. 2014

- Two key differences:
  - 1 level of parallelism (DF-Threads) vs. 2 (Codelets + TPs)
  - Each DF-Thread has its own private frame
  - All codelets belonging to a TP share the same TP frame (and data)
- DARTS maps each codelet to a DF-Thread, with a minimal DF-frame
- The TP frame shared by codelets siblings is allocated on the heap
- All codelets belonging to a TP are constrained to the same node

See Solinas et al. 2013; Giorgi et al. 2014; Zuckerman, Arteaga, et al. 2014

- Adding a codelet to the graph during execution triggers the call to `df_tschedule(&Fire,nb_deps,sizeof(Codelet*))`
- The `Fire` function is tasked to call the `Codelet::fire()` function and then clean up after using `df_destroy()`
- Threaded procedures are called using the `invoke<ThdProc>(parameters)` function:
  - Parameters are marshalled along with the TP type, and bundled within a DF-Thread
  - When firing, the DF-Thread allocates the TP on the heap, along with all of its codelets

See Solinas et al. 2013; Giorgi et al. 2014; Zuckerman, Arteaga, et al. 2014

All latencies were obtained using CACTI. We used COTSon's dynamic samplers to measure time (sample = 5M instructions).

	Private / Shared	Size	Number of Sets	Cache Line Size (Bytes)	Latency
L1D cache	private	16 KiB	4	64	2
L1I cache	private	32 KiB	4	64	2
L2U cache	private	64 KiB	4	64	5
L3U cache	shared	4 MiB	8	128	10

See Solinas et al. 2013; Giorgi et al. 2014; Zuckerman, Arteaga, et al. 2014

Running Fibonacci in DARTS – TERAFLUX

Strong Scaling

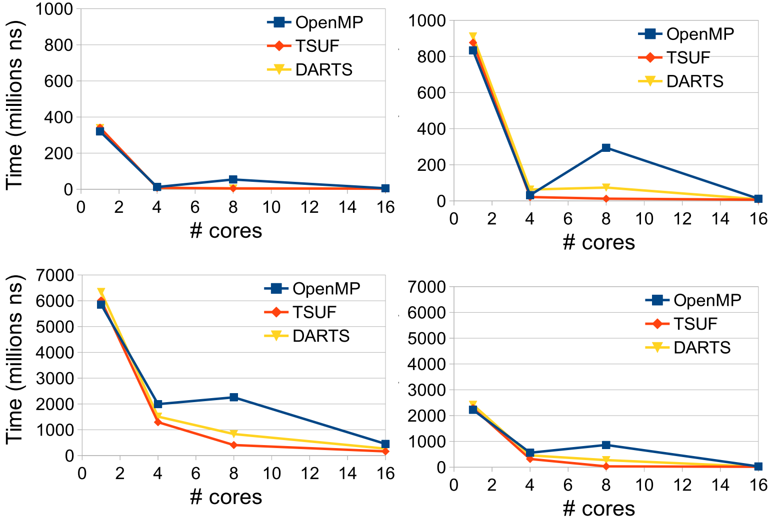


Figure : *Cutoff* $= 18$ – Fibonacci – $n = 36$ – Strong Scaling

Running Fibonacci in DARTS – TERAFLUX

Weak Scaling

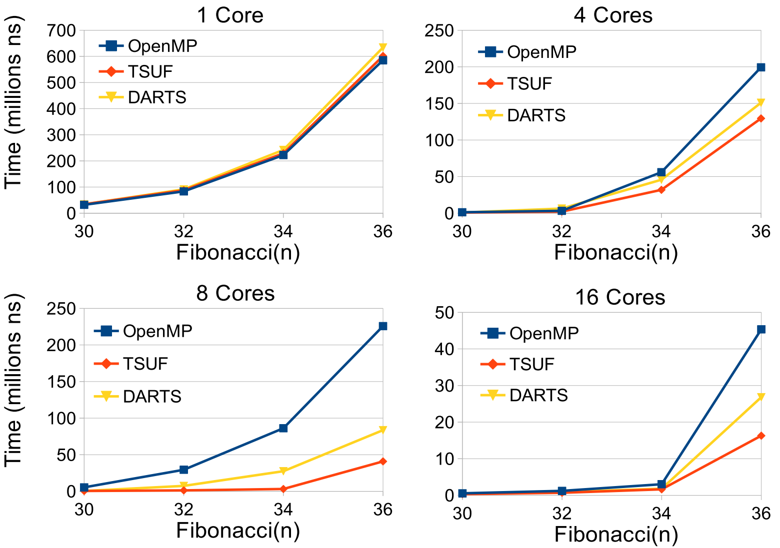


Figure : $Cutoff = 18$ – Fibonacci(n) – Weak Scaling

Running Merge Sort in DARTS – TERAFLUX

Strong Scaling

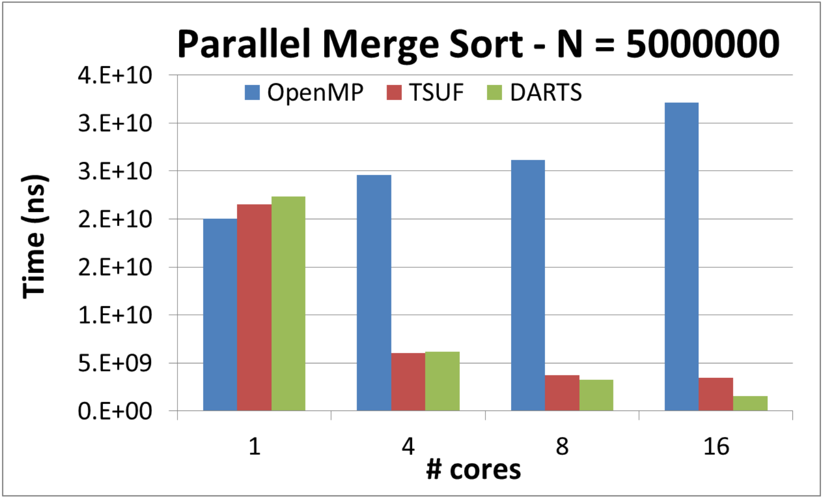


Figure : *Cutoff* $= 18$ – Merge Sort – $n = 5M$ elements – Strong Scaling

Running Merge Sort in DARTS – TERAFLUX

Weak Scaling

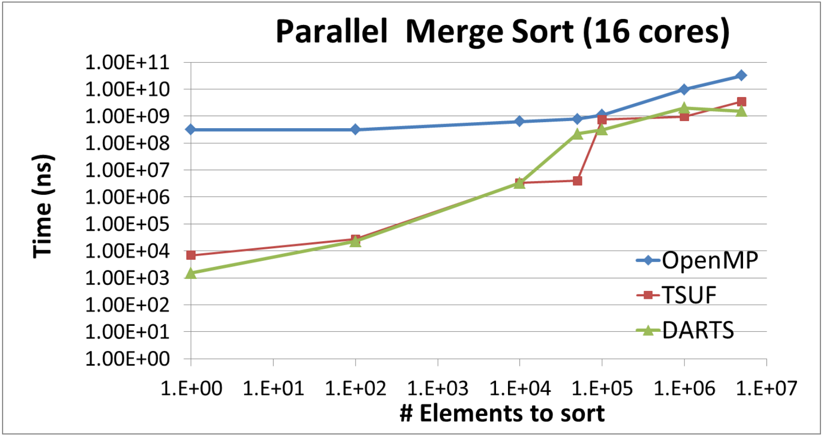


Figure : *Cutoff* = 10000 – Merge Sort(n) – Weak Scaling

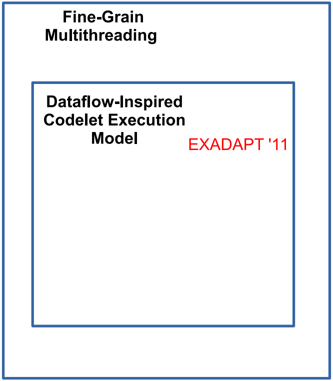
See Zuckerman, Arteaga, et al. 2014

Outline

The Future of Codelets

The Story So Far

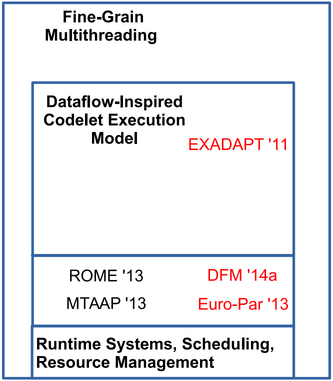
- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability
- Experimental results show that the Codelet Model can be competitive with current multicore environments
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores



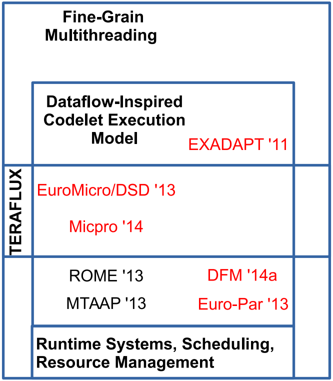
The Future of Codelets

The Story So Far

- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability
- Experimental results show that the Codelet Model can be competitive with current multicore environments
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores

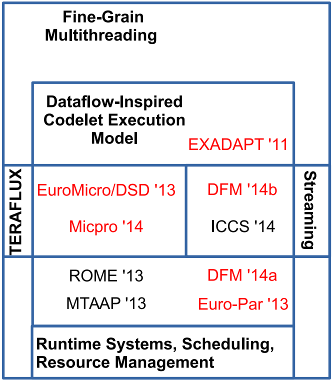


- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability
- Experimental results show that the Codelet Model can be competitive with current multicore environments
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores



The Future of Codelets
The Story So Far

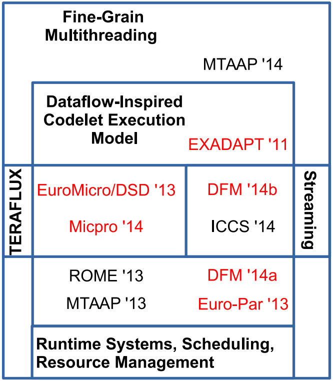
- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability
- Experimental results show that the Codelet Model can be competitive with current multicore environments
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores



The Future of Codelets

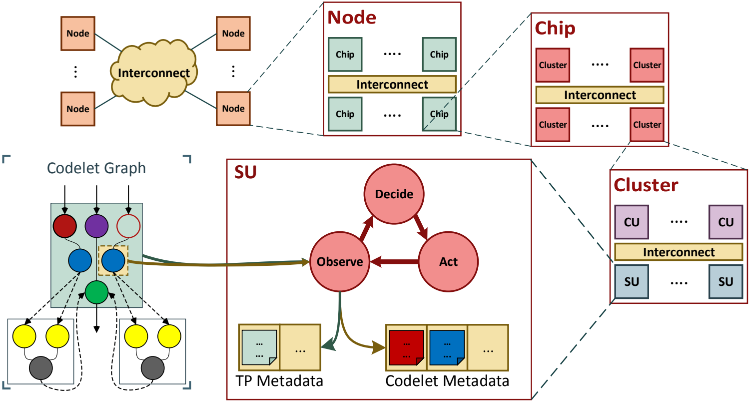
The Story So Far

- We proposed the codelet execution model to answer the need for scalability, performance, energy efficiency, fault-tolerance, and programmability
- Experimental results show that the Codelet Model can be competitive with current multicore environments
- With hardware support, the Codelet Model displays very high potential to scale to large numbers of cores



Extending the Codelet Model

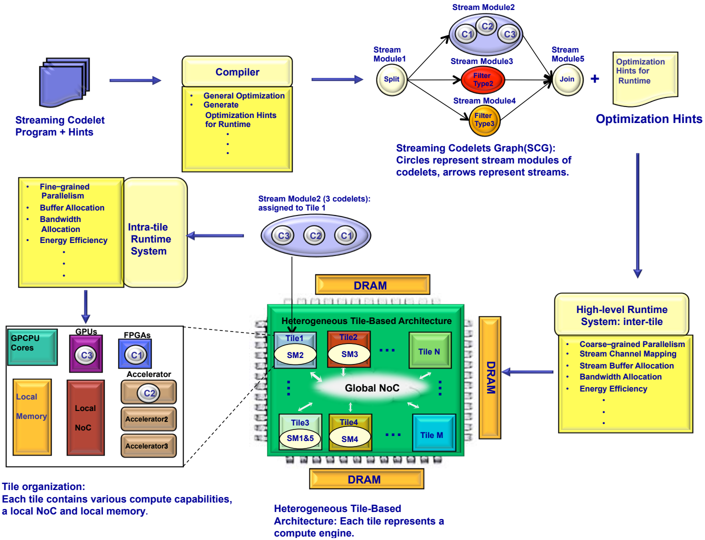
Extending Codelets to Self-Awareness



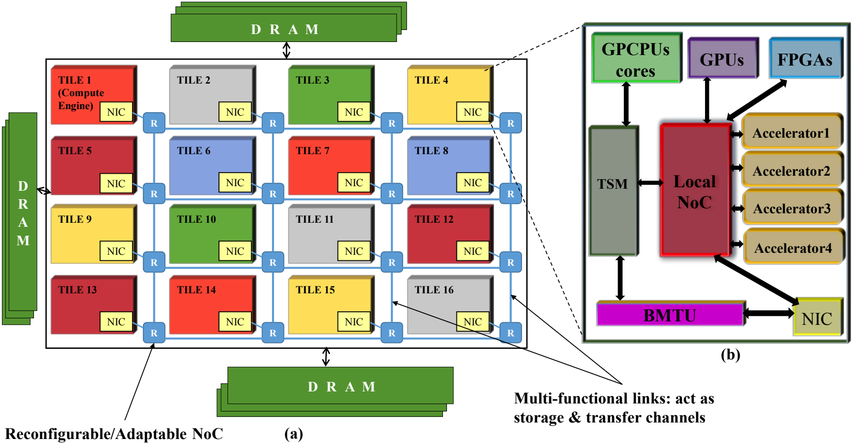
See Zuckerman, Landwehr, et al. 2014 for more details.

Extending the Codelet Model

Extending Codelets to Streams



See Zuckerman, Wei, et al. 2014 for more details.



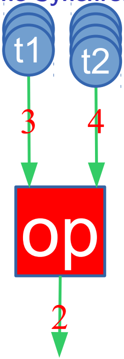
See Zuckerman, Wei, et al. 2014 for more details.

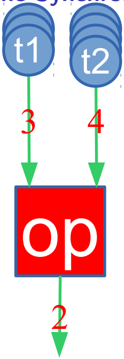
Past and present CAPSL members,

Brian	Lucas	Jaime	Arteaga
Joseph	Manzano	Chen	Chen
Daniel	Orozco	Elkin	Garcia
Robert	Pavel	Souad	Koliaï
Sergio	Pino	Aaron	Landwehr
Jürgen	Ributzka	Josh	Landwehr
Sunil	Shrestha	Kelly	Livingston
		Joshua	Suetterlein
Pouya	Fotouhi	Haitao	Wei
José	Monsalve	Yao	Wu

. . . And of course, Professor Gao!

The Synchronous Dataflow Model (Lee and Messerschmitt 1987)





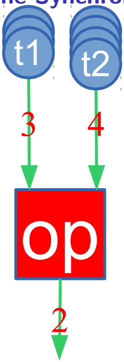
Synchronous Dataflow Actors

Components of a regular actor:

- Input arcs; each input arc i_l can contain a certain number $k_l i$ of tokens
- Output arcs; each output arc o'_l can contain a certain number $k'_{l'}$ of tokens
- The operation provided by the actor
- Tokens

A Short Introduction to Execution Models

The Synchronous Dataflow Model (Lee and Messerschmitt 1987)



Synchronous Dataflow Actors

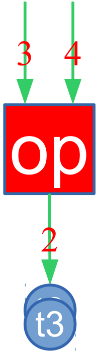
Components of a regular actor:

- Input arcs; each input arc i_l can contain a certain number $k_l i$ of tokens
- Output arcs; each output arc o'_l can contain a certain number $k'_{l'}$ of tokens
- The operation provided by the actor
- Tokens

Firing Rule: Synchronous Dataflow (SDF)

An actor may *fire* when:

- Each of its input arcs in the tuple $< i_0, i_1, \cdots, i_{k-1} >$ contains at least $< n_0, n_1, \cdots, n_{k-1} >$ tokens, and
- The number of slots available on each of the output arcs $< o_1, o_2, \cdots, o_{k'-1} >$ is sufficient to receive an additional count of n' tokens.



Synchronous Dataflow Actors

Components of a regular actor:

- Input arcs; each input arc i_l can contain a certain number $k_l i$ of tokens
- Output arcs; each output arc o'_l can contain a certain number $k'_{l'}$ of tokens
- The operation provided by the actor
- Tokens

Firing Rule: Synchronous Dataflow (SDF)

An actor may *fire* when:

- Each of its input arcs in the tuple $< i_0, i_1, \cdots, i_{k-1} >$ contains at least $< n_0, n_1, \cdots, n_{k-1} >$ tokens, and
- The number of slots available on each of the output arcs $< o_1, o_2, \cdots, o_{k'-1} >$ is sufficient to receive an additional count of n' tokens.

Other Dataflow Models

Dynamic Dataflow

- Allows for arbitrary recursions
- Relies on “color-matching:”
  - Each iteration is assigned a “color,”
  - An actor only fires if all tokens from the same color are present on its input arcs.
- Proved to provide maximum parallelism
- However: Color matching is *slow* (use of hash tables, . . .)

Macro-Dataflow

- Idea: Instead of relying on fine-grain, one-operation-at-a-time actors, let’s use a bunch of instructions/operations in sequence within the actor
- Still relies on inputs and outputs, but now the buffers may become much bigger, due to the amount of work and data required
- Offers a compromise to reduce the signaling overhead of fine-grain dataflow, token matching, *etc.*

See Watson and Gurd 1982; Arvind and Culler 1986; Papadopoulos and Culler 1990; Arvind and Gostelow 1982

Extending the Codelet Model

Extending Codelets to Self-Awareness

Future extreme-scale systems will most likely feature thousands of cores on a chip, and deep memory hierarchies. The Codelet Model was created with this in mind. However several problems still need to be tackled:

- Memory movements are expected to cost much more than computations in terms of energy consumption
- There will be a need for fine-grain resource management to target goals such as:
  - Maximum or average power envelope during computation required by the user
  - Degree of parallelism in the application declared by the user
  - Maximum acceptable temperature levels
  - ...
- We want to augment codelets and threaded procedures with meta-data which describe their resource usage
- A low-level runtime will then be able to make smart decisions based on static meta-data as well as updated data collected during the codelets executions
- The runtime will then be able to decide when to turn on/off parts of the manycore ships, when to rely on DVFS techniques, *etc.*

See Zuckerman, Landwehr, et al. 2014 for more details.

Dataflow naturally maps to streams. However, the *nature* of future extreme-scale manycore processors will most likely be widely different:

- Heterogeneity is already becoming a reality at the chip level
  - AMD released its first Fusion processor (CPU+GPU on the same chip) in 2013
  - Next-generation Intel "co-processors" will be on package with traditional multicore chips
  - Nvidia just recently produced an accelerator board which embed arm processors to deal with more control-heavy workloads
- We predict heterogeneity will be ingrained at a much deeper level in processors
- This is a great opportunity to do research in that direction — and in particular by targeting streams

The NSF just accepted to provide funding to explore this venue. A high-level view of our objectives was recently published. See Zuckerman, Wei, et al. 2014 for more details.

References I

Compiler Optimization, Performance Analysis, and Code Transformations in Multicore Systems

- Patrick Carribault et al. (2007). "Deep Jam: Conversion of Coarse-Grain Parallelism to Fine-Grain and Vector Parallelism." In: *J. Instruction-Level Parallelism* 9. URL: http://www.jilp.org/vol9/v9paper11.pdf

- Stéphane Zuckerman, Marc Pérache, and William Jalby (2008). "Fine Tuning Matrix Multiplications on Multicore." In: *High Performance Computing - HiPC 2008, 15th International Conference, Bangalore, India, December 17-20, 2008. Proceedings*. Ed. by P. Sadayappan et al. Vol. 5374. Lecture Notes in Computer Science. Springer, pp. 30–41. ISBN: 978-3-540-89893-1. DOI: 10.1007/978-3-540-89894-8_7. URL: http://dx.doi.org/10.1007/978-3-540-89894-8_7
- Souad Koliaï et al. (2009). "A Balanced Approach to Application Performance Tuning." In: *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009, Newark, DE, USA, October 8-10, 2009, Revised Selected Papers*. Ed. by Guang R. Gao et al. Vol. 5898. Lecture Notes in Computer Science. Springer, pp. 111–125. ISBN: 978-3-642-13373-2. DOI: 10.1007/978-3-642-13374-9_8. URL: http://dx.doi.org/10.1007/978-3-642-13374-9_8

References II
Compiler Optimization, Performance Analysis, and Code Transformations in Multicore Systems

- Benedetto Risio et al. (2009). "How to Accelerate an Application: a Practical Case Study in Combustion Modelling." In: *Parallel Computing: From Multicores and GPU's to Petascale, Proceedings of the conference ParCo 2009, 1-4 September 2009, Lyon, France*. Ed. by Barbara M. Chapman et al. Vol. 19. Advances in Parallel Computing. IOS Press, pp. 661–668. ISBN: 978-1-60750-529-7. DOI: 10.3233/978-1-60750-530-3-661. URL: http://dx.doi.org/10.3233/978-1-60750-530-3-661
- Andres Charif Rubial et al. (2009). "An Approach to Application Performance Tuning." In: *Parallel Computing: From Multicores and GPU's to Petascale, Proceedings of the conference ParCo 2009, 1-4 September 2009, Lyon, France*. Ed. by Barbara M. Chapman et al. Vol. 19. Advances in Parallel Computing. IOS Press, pp. 653–660. ISBN: 978-1-60750-529-7. DOI: 10.3233/978-1-60750-530-3-653. URL: http://dx.doi.org/10.3233/978-1-60750-530-3-653
- Stéphane Zuckerman and William Jalby (2010). "Tackling Cache-Line Stealing Effects Using Run-Time Adaptation." In: *Languages and Compilers for Parallel Computing - 23rd International Workshop, LCPC 2010, Houston, TX, USA, October 7-9, 2010. Revised Selected Papers*. Ed. by Keith D. Cooper, John M. Mellor-Crummey, and Vivek Sarkar. Vol. 6548. Lecture Notes in Computer Science. Springer, pp. 62–76. ISBN: 978-3-642-19594-5. DOI: 10.1007/978-3-642-19595-2_5. URL: http://dx.doi.org/10.1007/978-3-642-19595-2_5

References – Codelet Model I

Specification and Implementation

- Guang R. Gao, Joshua Suetterlein, and Stéphane Zuckerman (Apr. 2011). *Toward an Execution Model for Extreme-Scale Systems-Runnemede and Beyond*. Technical Memo 104. University of Delaware, 140 Evans Hall, Newark,DE 19716: Computer Architecture & Parallel Systems Laboratory, Electrical & Computer Engineering Departement
- Stéphane Zuckerman, Joshua Suetterlein, et al. (2011). "Using a "Codelet" Program Execution Model for Exascale Machines: Position Paper". In: *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*. EXADAPT '11. San Jose, California: ACM, pp. 64–69. ISBN: 978-1-4503-0708-6. DOI: 10.1145/2000417.2000424. URL: http://doi.acm.org/10.1145/2000417.2000424
- Joshua Suetterlein, Stéphane Zuckerman, and Guang R. Gao (2013). "An Implementation of the Codelet Model." In: *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*. Ed. by Felix Wolf, Bernd Mohr, and Dieter an Mey. Vol. 8097. Lecture Notes in Computer Science. Springer, pp. 633–644. ISBN: 978-3-642-40046-9. DOI: 10.1007/978-3-642-40047-6_63. URL: http://dx.doi.org/10.1007/978-3-642-40047-6_63

References – Codelet Model II

Specification and Implementation

- Haitao Wei et al. (2014). "A Dataflow Programming Language and its Compiler for Streaming Systems." In: *Proceedings of the International Conference on Computational Science, ICCS 2014, Cairns, Queensland, Australia, 10-12 June, 2014*. Ed. by David Abramson et al. Vol. 29. Procedia Computer Science. Elsevier, pp. 1289–1298. DOI: 10.1016/j.procs.2014.05.116. URL: http://dx.doi.org/10.1016/j.procs.2014.05.116
- Stéphane Zuckerman, Aaron Landwehr, et al. (2014). "Toward a Self-Aware Codelet Execution Model". In: *Proceedings of the Workshop on DataFlow Models for extreme-scale computing (DFM'14)*. Edmonton, AB, Canada
- Stéphane Zuckerman, Haitao Wei, et al. (2014). "A Holistic Dataflow-Inspired System Design". In: *Proceedings of the Workshop on DataFlow Models for extreme-scale computing (DFM'14)*. Edmonton, AB, Canada

References – TERAFLUX

Making Codelets converge with DF-Threads

- Marco Solinas et al. (2013). "The TERAFLUX Project: Exploiting the DataFlow Paradigm in Next Generation Teradevices." In: *2013 Euromicro Conference on Digital System Design, DSD 2013, Los Alamitos, CA, USA, September 4-6, 2013*. IEEE, pp. 272–279. DOI: 10.1109/DSD.2013.39. URL: http://dx.doi.org/10.1109/DSD.2013.39
- Roberto Giorgi et al. (2014). "TERAFLUX: Harnessing dataflow in next generation teradevices". In: *Microprocessors and Microsystems*, ISSN: 0141-9331. DOI: http://dx.doi.org/10.1016/j.micpro.2014.04.001. URL: http://www.sciencedirect.com/science/article/pii/S0141933114000490
- Stéphane Zuckerman, Jaime Arteaga, et al. (Apr. 2014). *D9.3 – Evaluation of the Codelet Runtime System on a Teradevice*. Deliverable 9.3. University of Delaware, 140 Evans Hall, Newark,DE 19716: Computer Architecture & Parallel Systems Laboratory, Electrical & Computer Engineering Departement

- Jaime Arteaga et al. (May 2014). "Position Paper: Locality-Driven Scheduling of Tasks for Data-Dependent Multithreading". In: *Workshop on Multi-Threaded Architectures and Applications (MTAAP 2014)*. Phoenix, USA
- Chen Chen et al. (2013). "Towards Memory-Load Balanced Fast Fourier Transformations in Fine-Grain Execution Models." In: *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*. IEEE, pp. 1607–1617. DOI: 10.1109/IPDPSW.2013.47. URL: http://dx.doi.org/10.1109/IPDPSW.2013.47

Other References I

- Jack B. Dennis, John B. Fosseen, and John P. Linderman (1972). "Data flow schemas". In: *International Sympoisum on Theoretical Programming, Novosibirsk, Russia, August 7-11, 1972, Proceedings*. Ed. by Andrei P. Ershov and V. A. Nepomniaschy. Vol. 5. Lecture Notes in Computer Science. Springer, pp. 187–216. ISBN: 3-540-06720-5. DOI: 10.1007/3-540-06720-5_15. URL: http://dx.doi.org/10.1007/3-540-06720-5_15
- Jack B. Dennis (1974). "First version of a data flow procedure language". In: *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*. Ed. by Bernard Robinet. Vol. 19. Lecture Notes in Computer Science. Springer, pp. 362–376. ISBN: 3-540-06859-7. DOI: 10.1007/3-540-06859-7_145. URL: http://dx.doi.org/10.1007/3-540-06859-7_145
- Jack B. Dennis and David Misunas (1974). "A Preliminary Architecture for a Basic Data Flow Processor". In: *Proceedings of the 2nd Annual Symposium on Computer Architecture, December 1974*. Ed. by Willis K. King and Oscar N. Garcia. ACM, pp. 126–132. DOI: 10.1145/642089.642111. URL: http://doi.acm.org/10.1145/642089.642111
- Arvind and Kim P. Gostelow (1982). "The U-Interpreter". In: *IEEE Computer* 15.2, pp. 42–49. DOI: 10.1109/MC.1982.1653940. URL: http://doi.ieeecomputersociety.org/10.1109/MC.1982.1653940
- Ian Watson and John R. Gurd (Feb. 1982). "A Practical Data Flow Computer". In: *Computer* 15.2, pp. 51–57. ISSN: 0018-9162. DOI: 10.1109/MC.1982.1653941

Other References II

- Arvind and David E. Culler (1986). "Dataflow Architectures". In: *Annual Review of Computer Science* 1.1, pp. 225–253. DOI: 10.1146/annurev.cs.01.060186.001301. URL: http://dx.doi.org/10.1146/annurev.cs.01.060186.001301
- E.A Lee and D.G. Messerschmitt (Sept. 1987). "Synchronous data flow". In: *Proceedings of the IEEE* 75.9, pp. 1235–1245. ISSN: 0018-9219. DOI: 10.1109/PROC.1987.13876
- Gregory M. Papadopoulos and David E. Culler (1990). "Monsoon: An Explicit Token-store Architecture". In: *Proceedings of the 17th Annual International Symposium on Computer Architecture*. ISCA '90. Seattle, Washington, USA: ACM, pp. 82–91. ISBN: 0-89791-366-3. DOI: 10.1145/325164.325117. URL: http://doi.acm.org/10.1145/325164.325117
- DARPA-BAA-10-37 (2010-2012). "UHPC: Ubiquitous High Performance Computing". In: Arlington VA, USA: DARPA
- Nicholas P Carter et al. (2013). "Runnemede: An Architecture for Ubiquitous High-Performance Computing". In: HPCA. Shenzhen, China
- Department of Energy (2012–2014). *X-Stack — Extreme Scale Software Stack*. URL: http://www.xstack.org