Towards Productive Parallel Programming

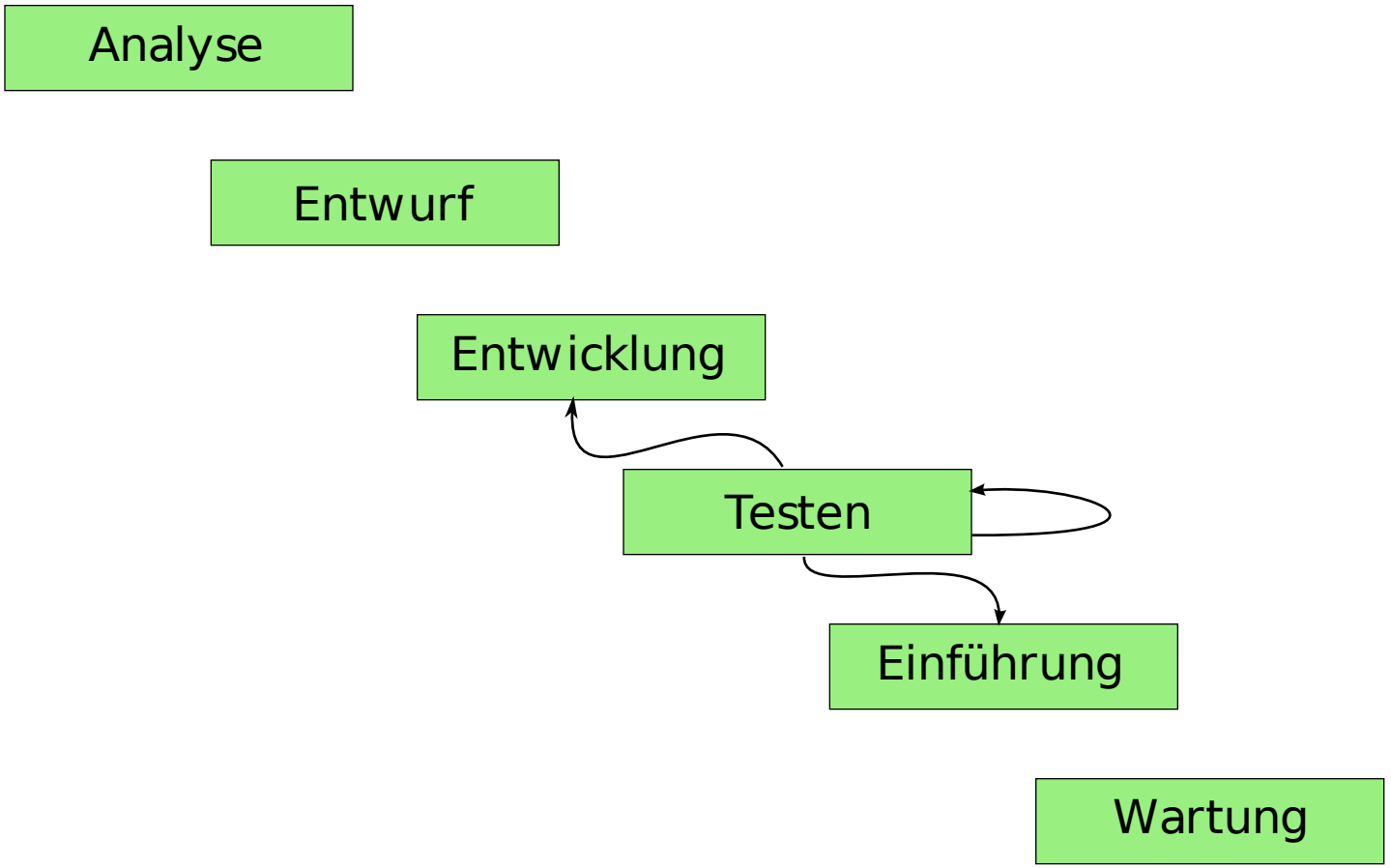
Christian Grothoff
christian@grothoff.org

University of Denver

Überblick

1. Kontrollflussorientierte Testverfahren
2. Productive Parallel Programming:
   - High-Performance Computing mit X10
   - Distributed Stream Processing mit DUP

Der Softwareentwicklungsprozess



Schlüsselfragen

- Warum schreiben wir einen Test?
- Was ist ein guter Test?
- Sind die vorhandenen Tests ausreichend?

Schlüsselfragen

- Warum schreiben wir einen Test?
- Was ist ein guter Test?
- Sind die vorhandenen Tests ausreichend?

  $\Rightarrow$ Was wird (noch) nicht getestet?

Modell-basiertes Testen

Grafen im Programm finden und durch Tests abdecken [5]:

- Anweisungen & Verzweigungen
- Funktionen & Funktionsaufrufe
- Kompontenten & Signale
- Zustand & Zustandsübergang

Kontrollflussorientierte Testverfahren

Strukturorientierte Testmethode bei der die Qualität einer Testserie nach bestimmten Kriterien mit Bezug auf den Kontrollfluss des Programs beurteilt:

- Knotenüberdeckung
- Kantenüberdeckung
- Pfadüberdeckung

Beispiel

```
int bitcount (unsigned char *bitmap,
              size_t length) {
  int ret = 0;
  for (size_t i=0;i<length*8;i++)
    if (bitmap[i/8] & (1<<(i%8)))
      ret++;
  return ret;
}
```

Kriterium: Knotenüberdeckung

- Jede **Anweisung** muss von den Tests mindestens einmal ausgeführt werden

  $\Rightarrow$ Kein "toter" Code

- `bitmap=`$\{$`255`$\}$, `length=1` genügt!

Beispiel: Knotenüberdeckung

```
int bitcount (unsigned char *bitmap,
              size_t length) {
  int ret = 0;
  for (size_t i=0;i<length*8;i++)
    if (bitmap[i/8] & (1<<(i%8)))
      ret++;
  return ret;
}
```

bitmap={255}, length=1

Kriterium: Kantenüberdeckung

- Jeder **Programmzweig** muss von den Tests mindestens einmal ausgeführt werden

  $\Rightarrow$ Beinhaltet Knotenüberdeckung

- `bitmap=`$\{$`254`$\}$, `length=1` genügt!

Beispiel: Kantenüberdeckung

```
int bitcount (unsigned char *bitmap,
              size_t length) {
  int ret = 0;
  for (size_t i=0;i<length*8;i++)
    if (bitmap[i/8] & (1<<(i%8)))
      ret++;
  return ret;
}
```

bitmap={254}, length=1

Kriterium: Pfadüberdeckung

- Jeder mögliche **Programmpfad** muss von den Tests mindestens einmal ausgeführt werden

  $\Rightarrow$ Beinhaltet Kantenüberdeckung

- Unmöglich in Programmen mit Schleifen

  $\Rightarrow$ Abschwächungen, zum Beispiel jede Schleife wird nur mindestens einmal übersprungen und einmal ausgeführt

- Zwei Aufrufe, zum Beispiel `bitmap={255}`, `length=0` und `bitmap={254}`, `length=1` notwendig

Beispiel: “Pfadüberdeckung”

```
int bitcount (unsigned char *bitmap,
              size_t length) {
  int ret = 0;
  for (size_t i=0;i<length*8;i++)
    if (bitmap[i/8] & (1<<(i&7)))
      ret++;
  return ret;
}
```

bitmap={255}, length=0

bitmap={254}, length=1

Werkzeuge: gcov & lcov

- Überdeckung für Testserien ist nicht offensichtlich

  $\Rightarrow$ Einsatz von Werkzeugen zur Beurteilung

- `gcov` — Überdeckungsbeurteilung für GCC
- `lcov` — Visualisierung (Zeilenüberdeckung)

Benutzung

1. Code muss mit `--coverage` von `gcc` übersetzt werden
2. `lcov --directory .  --zerocounters`
3. Testserie normal ausführen (z.B. `make check`)
4. `lcov --directory .  --capture -o app.info`
5. `mkdir -p doc/coverage`
6. `genhtml -o doc/coverage app.info`

Ausgabe

```
 1                :#include <assert.h>
 2                :#include <sys/types.h>
 3                :
 4                :int bitcount (unsigned char *bitmap,
 5              1 :                size_t length) {
 6              1 :  int ret = 0;
 7              9 :  for (size_t i=0;i<length*8;i++)
 8              8 :    if (bitmap[i/8] & (1<<(i&7)))
 9              0 :      ret++;
10              1 :  return ret;
11                :}
12                :
13                :int main(int argc, char** argv)
14              1 :{
15              1 :  unsigned char mymap[] = { 0 };
16              1 :  assert (0 == bitcount (mymap, 1));
17              1 :  return 0;
18                :}
```

Ausgabe

```
  1                 : #include <assert.h>
  2                 : #include <sys/types.h>
  3                 :
  4                 : int bitcount (unsigned char *bitmap,
  5               1 :                size_t length) {
  6               1 :   int ret = 0;
  7               9 :   for (size_t i=0;i<length*8;i++)
  8               8 :     if (bitmap[i/8] & (1<<(i&7)))
  9               7 :       ret++;
 10               1 :   return ret;
 11                 : }
 12                 :
 13                 : int main(int argc, char** argv)
 14               1 : {
 15               1 :   unsigned char mymap[] = { 254 };
 16               1 :   assert (7 == bitcount (mymap, 1));
 17               1 :   return 0;
 18                 : }
```

Ausgabe

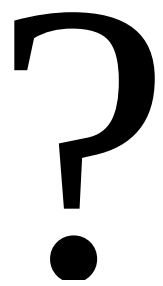
```
  1                 : #include <assert.h>
  2                 : #include <stdlib.h>
  3                 : #include <sys/types.h>
  4                 :
  5                 : int *bits2ints (unsigned char *bitmap,
  6               1 :                  size_t nbits) {
  7               1 :   int *ret = malloc(sizeof(int) * nbits);
  8               1 :   if (ret == NULL)
  9               0 :     return NULL;
 10               2 :   for (size_t i=0;i<nbits;i++)
 11               1 :     ret[i] = bitmap[i/8] & (1<<(i&7)) ? 1:0;
 12               1 :   return ret;
 13                 : }
 14                 :
 15                 : int main(int argc, char** argv)
 16               1 : {
 17               1 :   unsigned char mymap[] = { 254 };
 18               1 :   assert (NULL != bits2ints (mymap, 1));
 19               1 :   return 0;
 20                 : }
```

Grenzen Kontrollflussorientierter Testverfahren

- Skalierbarkeit (& Kosten)
- Grenzwerte (Ganzzahlüberlauf, ...)
- Parallele Datenverarbeitung:
  - Race Conditions
  - Deadlocks

  $\Rightarrow$ Statischer Kontrollfluss unerheblich!

Fragen

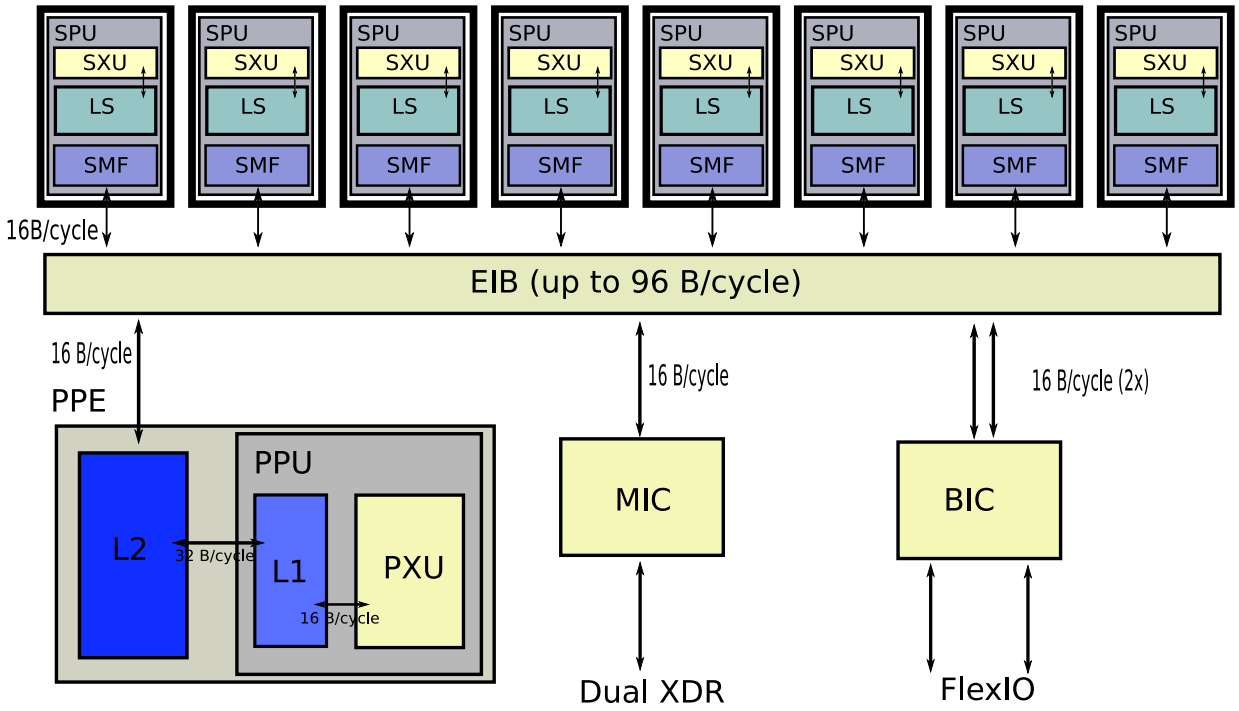


Parallel Computing is Mainstream

- **Desktop/Pentium Xeon:** hyperthreading, SMP
- **Notebook/Core Duo:** 2 cores
- **Playstation 3/Cell:** 9 processing units
- **Supercomputer/Blue Gene:** 128k processors

Programming these systems well is hard, even at 50% of peak!

Example: Cell Architecture



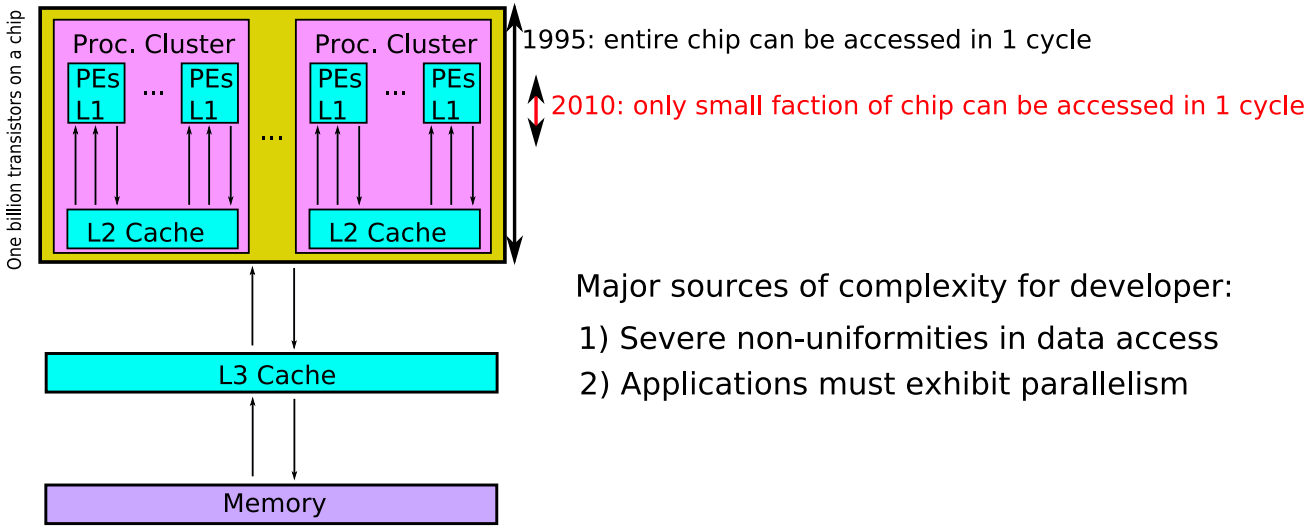
64-bit Power Architecture with VMX

How Much Faster?[1]

- Visualization: 146x
- Turbulence simulation: 17x
- Nbody simulation: 100x
- Molecular dynamics: 24x
- Gene sequence matching: 30x

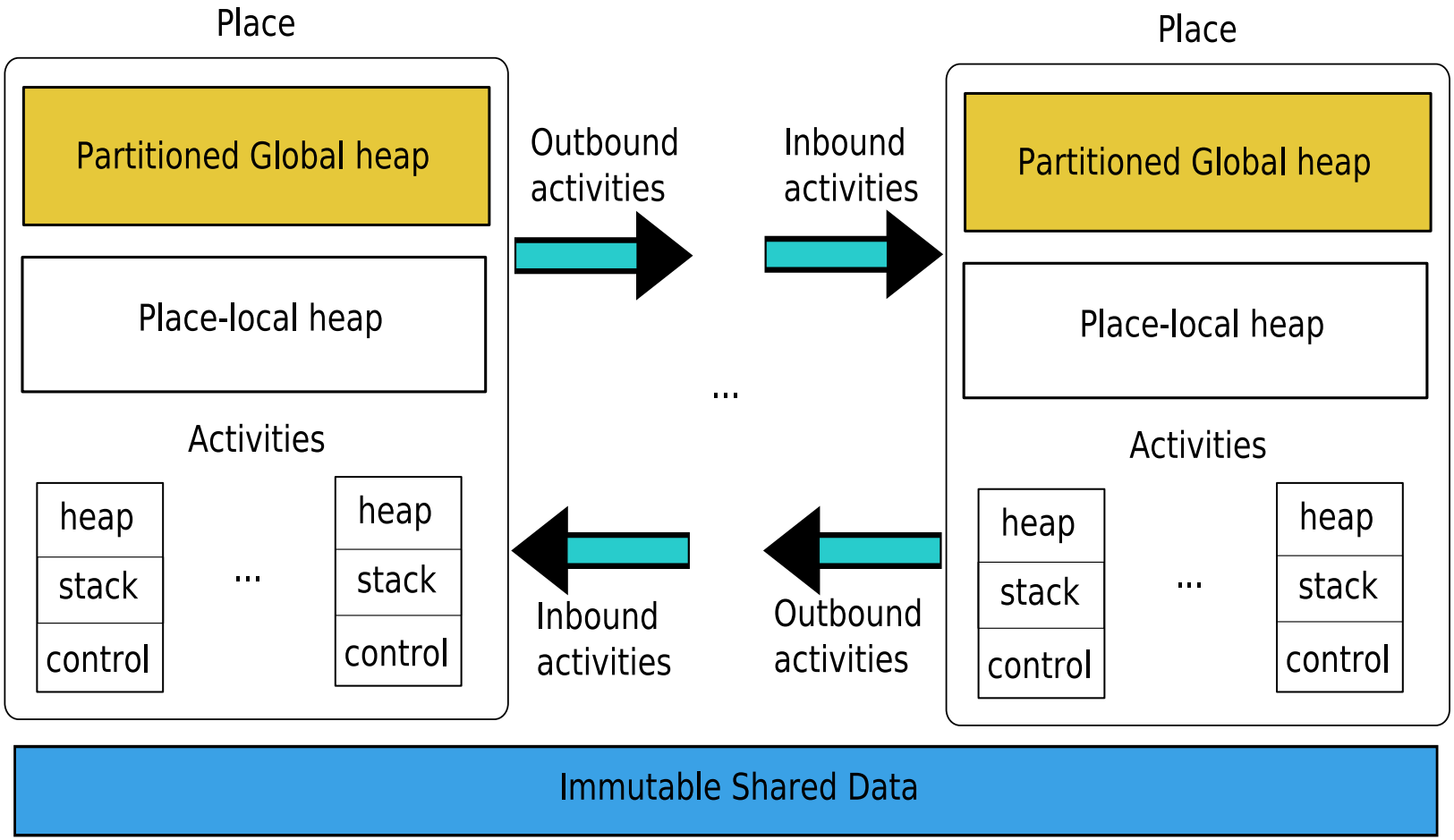
[1] According to http://www.nvidia.com/docs/IO/47904/VolumeI.pdf

Programming with Billions of Transistors



How long will application programmers be able to ignore this?

X10's Partitioned Global Address Space



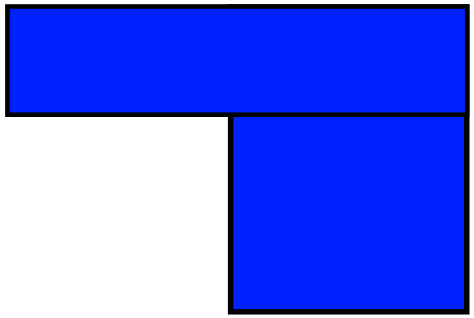
See [Charles et al, OOPSLA 2005] for details.

X10 Example Code

```
finish { // Intra-place parallelism
  int x = ..., y = ... ;
  async a.foo(x); // Initiate two activities at
  async b.bar(y); // place of parent activity
} // Wait for both activities to complete
finish { // Inter-place parallelism
  int x = ..., y = ... ;
  async (a) a.foo(x); // Execute at a's place
  async (b) b.bar(y); // Execute at b's place
}
```

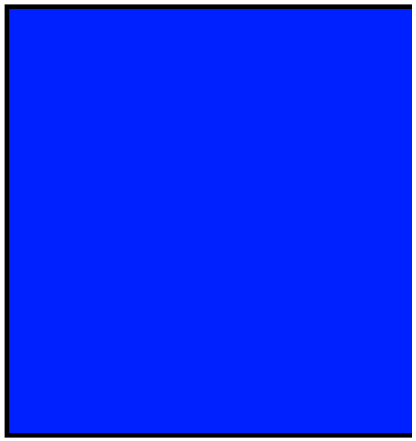
Regions vs. Intervals

```
region r = [0,0:40,10] | [20,10:40,30];
Array<int> a = new Array<int>(r);
```



```
for (p[i,j] : a.reg)
  a[p]++;
```

```
int[][] a = new int[40][40];
```



```
for (int i=0;i<40;i++)
  for (int j=0;j<40;j++)
    a[i][j]++;
```

ZPL, Titanium, Chapel and X10 all feature regions.

Regions in X10

- regions are first-class values in X10
- specify shape of and computations over arrays
- distributions are mappings of indices to places
- X10 features a rich region and distribution algebra

X10 Example Code

```
Array<int> A
  = new Array<int>([0,0:M,N], 0);
Array<int> B
  = new Array<int>(dist.block([0,0:M,N]), 0);

for     (p[i,j] : A.reg ) A[p] = A[j,i];
foreach (p      : A.reg ) A[p] = A[p]-1;
ateach  (p      : B.dist) B[p] = h(B[p]);
```

The trivial solution: dynamic checks

The simple approach is to copy from Java:

- `NullPointerException`
- `ArrayIndexOutOfBoundsException`
- `BadPlaceException`

Cost of dynamic checks

Benchmark	Total checks	runtime (s)	checks/s
Crypt [7]	3800520	15.6	245194
MolDyn [7]	308416	12.9	23908
RayTracer [7]	900	3.8	236
Conjugate gradient [2]	830899147	355.0	2340560

Experiments with the prototype show that for some benchmarks checks can cost a factor of 3 in performance!

Goal of the X10 Work

Design a **type system** to ensure memory safety for:

- systems with distributed memory
- languages with an array sublanguage

with intuitive and expressive annotations.

$\Rightarrow$ static safety, performance, self-documenting code

Relation to Prior Work

- X10's array sublanguage is quite expressive

  $\Rightarrow$ [Xi, 1998]'s DML(ILP) is not expressive enough

- [Liblit et al, 2003] only distinguishes global vs. local

  $\Rightarrow$ X10 needs support for multiple places

Our Approach

Uses **dependent types** that exploit the region algebra to eliminate dynamic checks:

- Add value properties to classes:

```
class Point (x:int, y:int) { ... }
```

- Allow types to be constrained using expressions over properties:

```
type X [Generics] (Properties) { Guard }
```

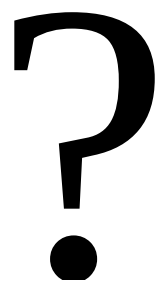
- Example: `C{self.loc==p}` or shorter `C!p`.

Summary

- Designed and prototyped object-oriented language for high-performance, high-productivity parallel computing
- Formalized type system to eliminate dynamic checks and statically guarantee safety

Details in "Constrained Types for Object-Oriented Languages" available at `http://grothoff.org/christian/oopsla2008.pdf`

Questions



The Problem: Developing Parallel Stream Applications

- Developers know how to write sequential code
- Parallel programing is error-prone
- High-performance parallel programming is really hard
- With GPUs for $4,000, we could have 2,600 cores...

  $\Rightarrow$ Developers much more expensive than hardware

X10 vs. the DUP System[2]

X10

10x faster, 10x as productive in 10 years for BlueGene

DUP

$\frac{1}{2}$ the speed, 10x as productive in 10 months for POSIX

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

A Blast from the Past: CMS Pipelines

- Similar to UNIX pipes
- Sligthly different syntax
- NEW: multistream pipelines

See also: CMS Pipelines User's Guide [11]

Example: CMS Pipeline

```
Pipe < INPUT FILE A % input is a stage!
|   drop 4           % like ‘‘eat 4’’
| sort 34-36         % sort by colums 34-36

| > OUTPUT FILE B    % output is a stage!
```

Example: CMS Multistream Pipeline

```
Pipe < INPUT FILE A
| d:drop 4   % label stage ‘‘d’’
| sort 34-36 %
| i:faninany % label stage  ‘‘i’’
| > OUTPUT FILE B
?            % end of primary pipeline
d:           % take 2nd output of ‘‘d’’
| i:         % make it the 2nd input of ‘‘i’’
```

Limitations of CMS Pipeines

- Sequential execution on one CPU, no parallelism
- Only available on CMS and z/OS
- Record-oriented (CMS is a mainframe OS)

... but these are easy to address!

Our Solution: DUP ≡ Distributed Multi-Stream Pipelines

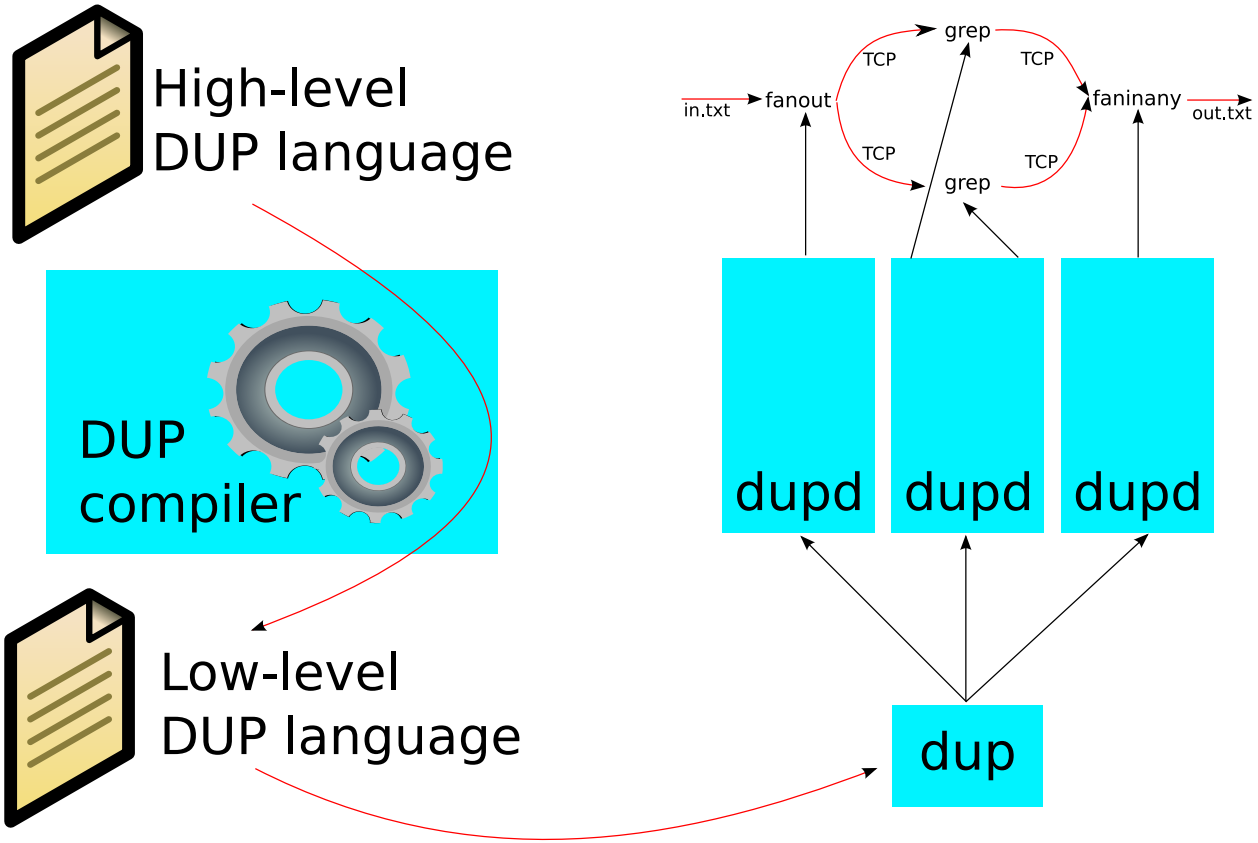
- Computation composed of stages in a flow-graph
- Stages run as individual processes in parallel
- Stages can have any number of inputs and outputs
- DUP used to connect stages
- DUP provides stages for common problems

⇒ Simple stream-oriented parallel programming model that also guides developers towards modular design

Example: DUP “Assembly”

```
dup <<EOF
drop@localhost[0<file.a,1|sort:0,3|merg:3] $ drop 4  ;
sort@localhost[1|merge:0]                  $ sort    ;
merg@localhost[1>file.b]                   $ faninany;
EOF
```

DUP Architecture



Vision: DUP High-level Language

```
import duplib;

$in = read("file.a");
($body, $head) = drop ($in, "4");
write (faninany (sort ($body),
                 $head),
        "file.b");
```

DUP Limitations

- Stages communicate via streams

  $\Rightarrow$ Computation must be stream-oriented

- Stages run in parallel, internals are up to the stage

  $\Rightarrow$ DUP parallelism limited by stages

DUP Application Domains

- Genome sequence processing
- Discrete event simulation
- Intrusion Detection
- Video conferencing
- Event surveillance
- System administration
- ...

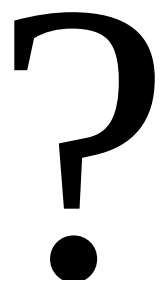
Related Work

- InfoSphere Streams [1] & Dryad [12]
- StreamFlex [15] & StreamIt [16]
- Kahn Process Networks [13]
- Linda [8]

Future Work

- High-level DUP programming language (will be an aspect-oriented coordination mini-language)
- Develop more filters/stages and applications
- Type systems for streams (see also: SPADE [10])
- Add common features of distributed systems [3, 4] while maintaining **simplicity**, **portability** and **language independence**

Questions



Copyright

References

[1] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *Workshop on Data Mining Standards, Services and Platforms (DM-SPP)*, 2006.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Shreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The nas parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[3] Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Eddie Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, and Stan Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[4] Magdalena Balazinska, Hari Balakrishnan, Samuel R. Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Trans. Database Syst.*, 33(1):1–44, 2008.

[5] Boris Beizer. *Software Testing Techniques*. Number 978-1850328803. International Thomson Computer Press, 2 edition, June 2000.

[6] Ian Buck. Gpu computing: Programming a massively parallel processor. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2007.

[7] J. M. Bull, L.A. Smith, D. S. Henty, and R. A. Davey. A benchmark suite for high performance java. *Concurrency: Practice and Experience*, 12(6):375–388, August 1999.

[8] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[9] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vivek Sarkar, and Christoph Von Praun. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 519–538. ACM SIGPLAN, 2005.

[10] Martin Hirzel, Henrique Andrade, Bugra Gedik, Vibhore Kumar, Giuliano Losa, Robert Soule, and Kun-Lung-Wu. Spade language specification. Technical report, IBM Research, March 2009.

[11] IBM. *CMS Pipelines User's Guide*. IBM Corp., http://publibz.boulder.ibm.com/epubs/pdf/hcsh1b10.pdf, version 5 release 2 edition, Dec 2005.

[12] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisabon, Portugal, March 2007.

[13] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing*, (74):993–998, 1974.

[14] Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *Proceedings of POPL'00, 27nd Annual SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 199–213, 2000.

[15] Jesper H. Spring, Jean Privat, Rachid Guerraoui, and Jan Vitek. Streamflex: high-throughput stream programming in java. *SIGPLAN Not.*, 42(10):211–228, 2007.

[16] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[17] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, September 1998.