Languages and Compilers for Productivity and Efficiency

Albert Cohen

with contributions from Riyadh Baghdadi, Léonard Gérard, Tobias Grosser, Adrien Guatto, Nhat Minh Lê, Feng Li, Antoniu Pop, Jean-Yves Vet, Sven Verdoolaege, and more

INRIA and École Normale Supérieure, Paris
`http://www.di.ens.fr/ParkasTeam.html`

Workshop on Virtual Machines and Multicore Architectures, September 27–28, 2012

Parallel Hardware: Where Are We Now?

DDR3-2133 SDRAM

Latency: $\mathbf{10.3}$ ns

Memory bandwidth: $\mathbf{17.6}$ GB/s

4-core 2GHz ARM Cortex A15 – 4W

Compute bandwidth: $\mathbf{2} \times \mathbf{4}$ threads $\times \mathbf{1}$ NEON unit $\times \mathbf{16}$ bytes $\times \mathbf{2}$ GHz $= \mathbf{256}$ GB/s

8-core 3GHz AMD Opteron Interlagos – 90W

Compute bandwidth: $\mathbf{2} \times \mathbf{8}$ threads $\times \mathbf{2}$ SSE units $\times \mathbf{16}$ bytes $\times \mathbf{3}$ GHz $= \mathbf{1536}$ GB/s

Memory bandwidth: $\mathbf{17.6}$ GB/s

256-core 400MHz Kalray MPPA – 5-10W?

Compute bandwidth: $\mathbf{2} \times \mathbf{256}$ threads $\times \mathbf{2}$ words $\times \mathbf{4}$ bytes $\times \mathbf{400}$ MHz $= \mathbf{1638.4}$ GB/s

1536-core 1.006GHz NVIDIA Kepler – 200-300W

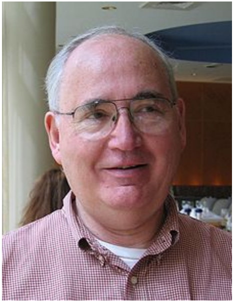
Compute bandwidth: $\mathbf{2} \times \mathbf{1536}$ threads $\times \mathbf{1}$ float $\times \mathbf{4}$ bytes $\times \mathbf{1.006}$ GHz $= \mathbf{12361.6}$ GB/s

Memory bandwidth: $\mathbf{190}$ GB/s

What Should We Do About It?

- **What are the essential semantic requirements for source programs?**
- **Should programmers care**
  About parallelism?
  About the memory and power walls?
  Which programmers?
- **What role for the software stack?**
  Compilers
  Runtime systems
  Libraries, library generators
  Auto-tuning, dynamic optimization
  Operating system, virtual machine monitor

What Foundations for Parallel Programming?



Domain theory of recursive functions: denotational semantics of a program as the least fixpoint of a system of equations over continuous functions

Dana Scott (1932–), Turing Award

Kahn process networks: system of equations over continuous functions on infinite streams (denotational); or processes communicating over infinite FIFOs with blocking reads (operational)

+ Function and parallel composition
+ Deterministic by construction
− Concurrent data structures, in-place operations missing
− How to "run" a Kahn process network efficiently?

Gilles Kahn (1946–2006), President and CEO of INRIA

1. Task Models

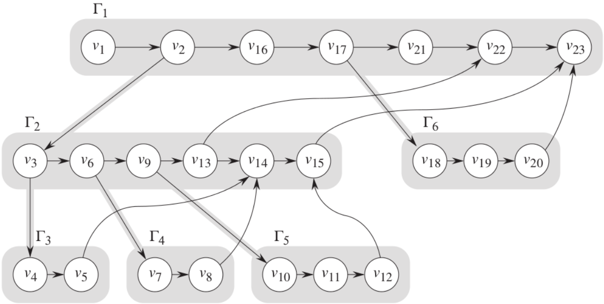
The Cilk Project

- C dialect for dynamic multithreaded applications
- Developed since 1994 at MIT in the group of Charles Leiserson
  `http://supertech.csail.mit.edu/cilk`
  Now part of Intel Parallel Studio (and TBB, ArBB)
- Tasks are (nested) coroutines
- Two keywords:
  - `retval =` *spawn* `function(args)` to indicate that the function call and its continuation may execute concurrently
  - *sync* to implement a join operation, waiting for all child tasks of the current task

```
cilk int fib (int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
    x = spawn fib (n-1);
    y = spawn fib (n-2);
    sync;
    return (x+y);
  }
}
```

Cilk Properties

Cilk programs are canonically sequentialized with the elision of the special keywords
$\rightarrow$ Depth-first execution of the task tree by a single-thread



$\rightarrow$ As a corollary, all inputs to a task are available at the task creation point
$\rightarrow$ A property called strictness (some relation to strictness in functional languages)

$\rightarrow$ Lots of benefits: absence of deadlocks, sequentialization/compilation of parallelism, faster/lighter runtime...

Work-Stealing, Lock-Free Deque

State-of-the-art implementation: David Chase and Yossi Lev 2005

- Uses a wrap-around buffer with automatic resizing
- No atomic compare-and-swap in the common case
- On x86, only needs one fence for each task

```
int take () {
  long b = bottom - 1;
  item_t *q = deque;
  bottom = b;
  MFENCE;
  long t = top;
  if (b < t) {
    bottom = t;
    return EMPTY;
  }
  int task = q->buf[b%q->size];
  if (b > t)
    return task;
  if (!atomic_cas(&top, t, t+1))
    return EMPTY;
  bottom = t + 1;
  return task;
}
```

```
void push (int task) {
  long b = bottom;
  long t = top;
  item_t *q = deque;
  if (b - t > q->size - 1)
    expand();
  q->buf[b%q->size] = task;
  bottom = b + 1;
}
```

```
void steal (int task,
              item_t *remote_deque) {
  long t = top;
  long b = bottom;
  item_t *q = remote_deque;
  if (t >= b)
    return EMPTY;
  int task = q->buf[t%q->size];
  if (!atomic_cas(&top, t, t+1))
    return ABORT;
  return task;
}
```

Work-Stealing on a Relaxed Memory Model

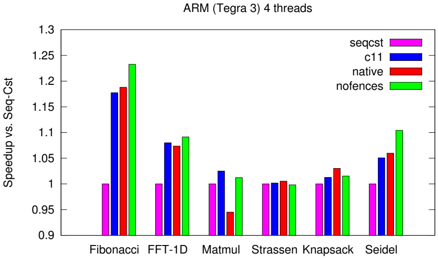
Portable C11 implementation
(formal proof of POWER/ARM version available, easily adaptable to C11)

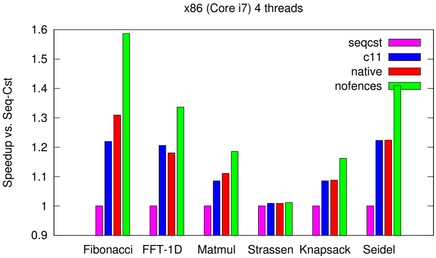
```
int take(deque_t *deque) {
  long b = load_explicit(&deque->bottom, relaxed) - 1;
  array_t *q = load_explicit(&deque->array, relaxed);
  store_explicit(&deque->bottom, b, relaxed);
  thread_fence(seq_cst);
  long t = load_explicit(&deque->top, relaxed);
  if (b < t) {
    store_explicit(&deque->bottom, b + 1, relaxed);
    return EMPTY;
  }
  int task = load_explicit(&q->buffer[b%q->size], relaxed);
  if (b > t)
    return task;
  if (!compare_exchange_strong_explicit(&deque->top,
      &t, t + 1, seq_cst, relaxed))
    task = NULL;
  store_explicit(&deque->bottom, b + 1, relaxed);
  return task;
}
```

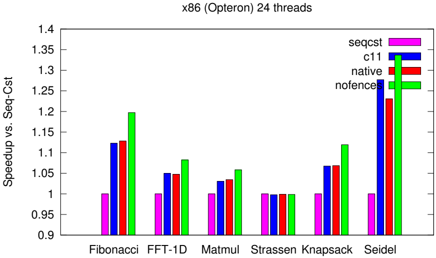
```
int steal(deque_t *remote_deque) {
  long t = load_explicit(&remote_deque->top, acquire);
  thread_fence(seq_cst);
  long b = load_explicit(&remote_deque->bottom, acquire);
  if (t >= b)
    return EMPTY;
  array_t *q = load_explicit(&remote_deque->array, relaxed);
  int task = load_explicit(&q->buffer[t%q->size], relaxed);
  if (!compare_exchange_weak_explicit(&remote_deque->top,
      &t, t + 1, seq_cst, relaxed))
    return ABORT;
  return task;
}
```

```
void push(deque_t *deque, int task) {
  long b = load_explicit(&deque->bottom, relaxed);
  long t = load_explicit(&deque->top, acquire);
  array_t *q = load_explicit(&deque->array, relaxed);
  if (b - t > q->size - 1)
    resize(deque);
  store_explicit(&q->buffer[b%q->size], task, relaxed);
  thread_fence(release);
  store_explicit(&deque->bottom, b + 1, relaxed);
}
```

Work-Stealing on a Relaxed Memory Model







Modern Work-Stealing Implementations and Task Models

Hierarchical, heterogeneous, distributed workstealing, accelerators

StarPU project in Bordeaux, now underlying the MAGMA and PLASMA libraries
`http://runtime.bordeaux.inria.fr/StarPU`

StarSs project in Barcelona, with OMPSs framework and Nanos runtime
`http://pm.bsc.es/ompss`

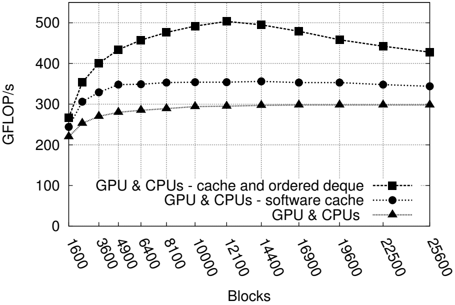
KAAPI project in Grenoble
`http://moais.imag.fr/membres/thierry.gautier/TG/home_page.html`

2. Multigranularity Scheduling With a Software Cache

Multigranularity Scheduling: Sparse LU

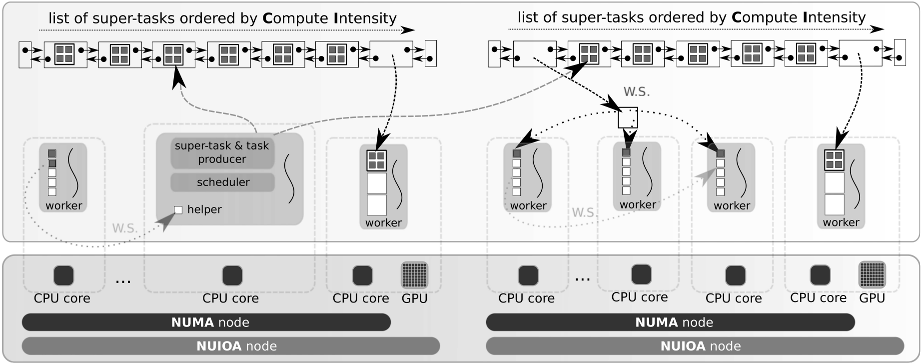
The need for hybrid CPU+GPU execution

Work-stealing for super-tasks: CPU+GPU
Work-stealing for tasks: CPU cores
Software cache with compute/reuse policy



Platform: 2 Opteron Magny-Cours (24 cores), 2 GTX 470 (Fermi)

Multigranularity Scheduling: Runtime System



Multigranularity Scheduling: PN application (CEA DAM)

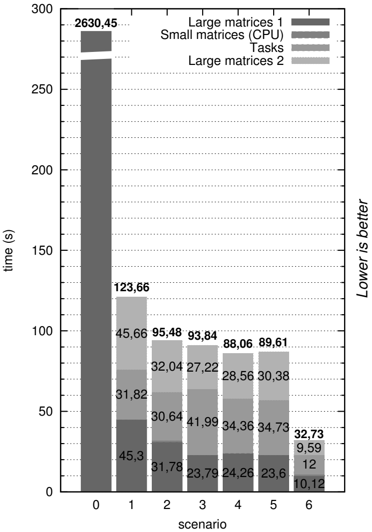
Effect of software cache replacement policy:
compute intensity vs. data intensity

Best strategy: hybrid

PN: 1536×1536 mesh, 36 iterations
Parallelization scheme for `numerical_flux`

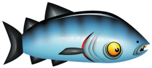
	Large GEMM 1	Small GEMM	Tasks	Large GEMM 2
0	sequential	sequential	sequential	sequential
1	CPU	CPU	CPU	CPU
2	CPU+GPU compute	CPU	CPU	CPU+GPU compute
3	CPU+GPU compute	CPU compute	CPU+GPU compute	CPU+GPU compute
4	CPU+GPU compute	CPU compute	CPU+GPU *data*	CPU+GPU compute
5	CPU+GPU compute	CPU compute	CPU+GPU *data*	CPU+GPU *data*
6	CPU+GPU no transfer	CPU	CPU+GPU no transfer	CPU+GPU no transfer

Platform: 2 Xeon Nehalem E5620 (16 cores), 2 Tesla 2090 (Fermi)



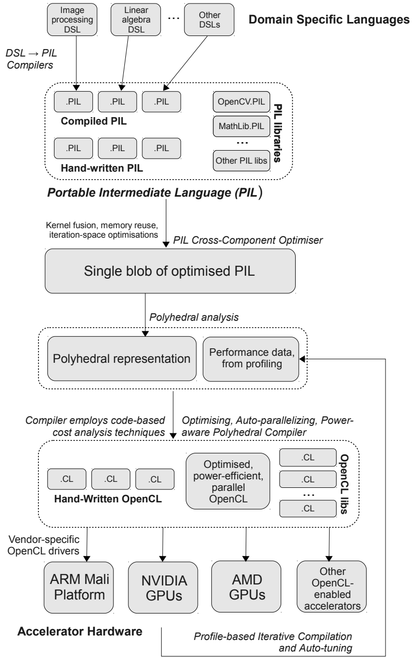
3. Generating Host and Kernel Code

CARP EU Project



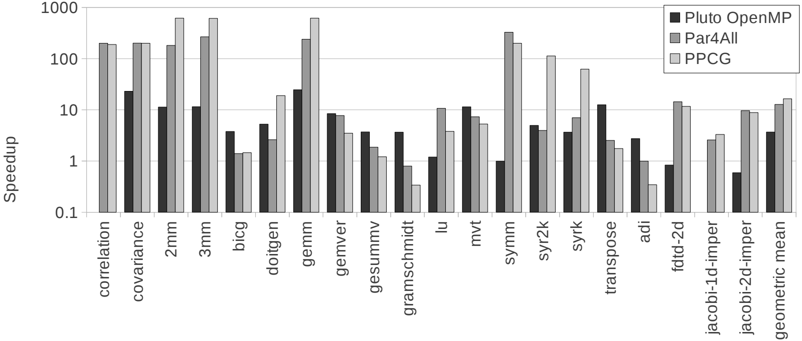
w/ ARM, RealEyes, Rightware, Monoidics, Imperial College, RWTH Aachen, U. Twente

- Compiler construction for DSLs: support for parallelization, vectorization, loop transformation...
- Reconcile advanced loop nest optimizations, software engineering practices, and formal verification methods

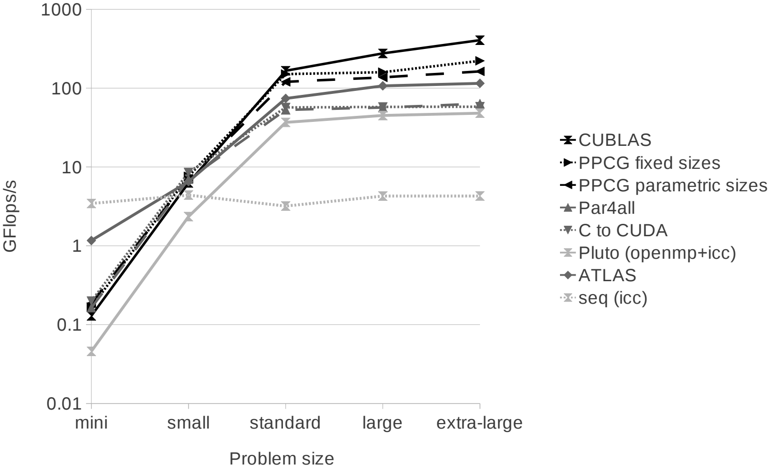


Polyhedral Compilation for NVIDIA Fermi: Polybench 3.1

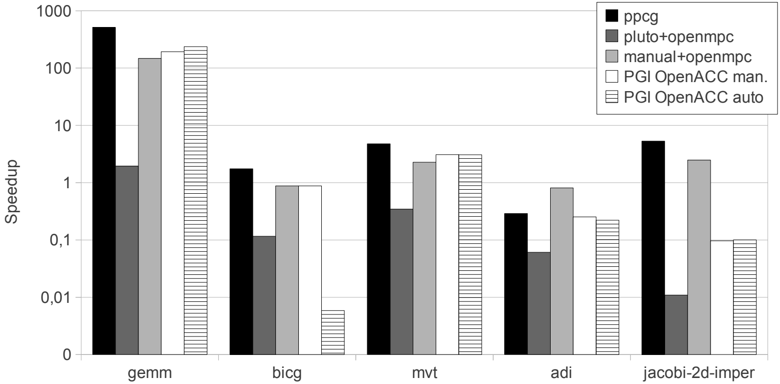
PPCG: `http://freecode.com/projects/ppcg`



Polyhedral Compilation for NVIDIA Fermi: GEMM



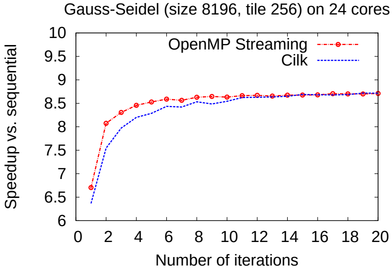
Polyhedral Compilation for NVIDIA Fermi: GEMM vs. Difficult Cases

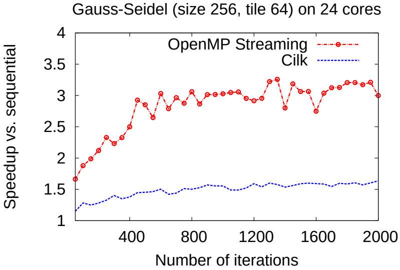


4. Dynamic Data Flow

Functional Determinism and Dependences

- Cilk only implements joins/barriers
- Cilk's tasks are immediately ready (price to pay for strictness)
- The schedule is over-constrained: detrimental to scalability and load balancing
- Motivation for a more expressive, data-flow task model





- How to implement Kahn networks with coroutines and a work-stealing scheduler?
- How to exented a scheduling algorithm to deal with dependent tasks?

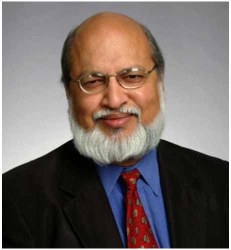
Data-Flow

Jack Dennis (1931–), Turing Award
Arvind, Culler, Iannuci, Nikhil, Pingali, Gao et al. (MIT)
Ian Watson, John Gurd (Manchester)
Motivation: hardware data-flow architectures

Goal: data is always local
when a task is scheduled/activated

Efficient use of local memories

Run-to-completion coroutines
→ linear stacks are sufficient



- Tasks/coroutines are called data-flow threads
- Activation records of (dependent) tasks are called data-flow frames
  - Explicitly managed by the scheduler in a dedicated heap structure
  - Frame allocation at thread creation, deallocation at termination
- Question: data-driven/feed-forward style with a synchronization counter (SC) vs. tag-based data flow with associative map?
  - Data-driven: remote writes to frames of consumer threads, decrement SC
  - Tag-based: tag every write, match consumer with ready tags

Sample Data-Driven Execution Primitives

Source: inspired from the DTA architecture, from Krishna Kavi and Roberto Giorgi

- `void *tcreate(void (*func)(), int sc, int size);`
  Allocates a new frame, sets the function pointer and initial SC
- `void tdecrease(void *fp);`
  Each call to tdecrease increments a thread-local counter to cache locally the value to be decremented on a given consumer thread
- `tend()`
  Atomically subtracts each thread-local counter from the corresponding dependent thread's SC; when any SC reaches 0, it inserts the corresponding thread to the ready queue of the current worker thread; deallocates the frame of the terminating thread and returns
- `tgetcfp()`
  Retrieve the current frame pointer from the thread-local storage area of the worker thread

Task-Parallelization of Basic Blocks

```
int caller()
{
BB1
  a1 = ...;

BB2
  d5 = a1*a1;

BB2
  ... = d5;
}
```

→

```
void caller.th2()
{
  fp2 = tgetcfp()

  a1 = fp2->a1;
  fp3 = fp2->fp3;
  d5 = a1*a1;
  fp3->d5 = d5;

  tdecrease(fp3);
}
```

Input arguments and pointers to the frames of the dependent frame are collected from the def-use chains (SSA form in compilers for imperative languages)

Modular Task-Parallelization of Function Calls

```
void caller.th2()
{
  fp2 = get_cfp()

  a1 = fp2->a1;
  fp3 = fp2->fp3;
  d5 = a1*a1;
  fp3->d5 = d5;

  tdecrease(fp3);
}
```

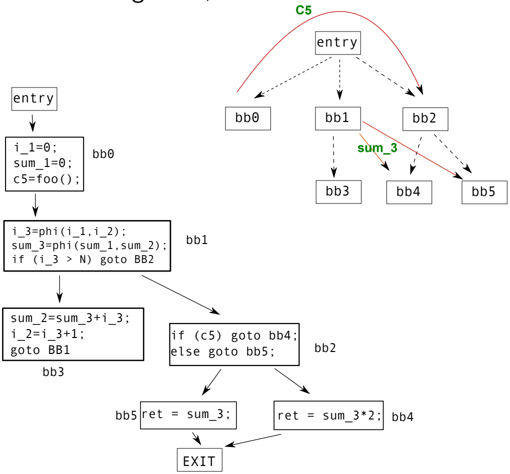
$\rightarrow$

```
int caller() {
  ...
  ret = callee(arg1);
  ...
  ... = ret;
}

void caller.bb.1() {
  // creation point
  fp_callee.entry = tcreate(callee.entry, sc, ...);
  fp_callee.return = tcreate(callee.return, sc, ...);
  fp_callee.entry->arg1 = arg1;
  fp_callee.entry->ret_addr = &fp_callee.return->ret;
  fp_callee.entry->ret_fp = fp_callee.return;
}
```

General Conversion of Control Flow to Data Flow

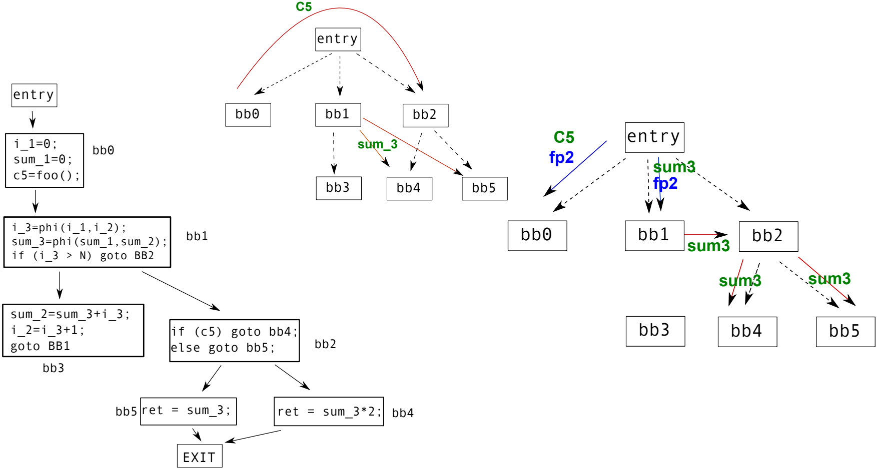
Problem: in general, the consumer is not known at production time



- When creating the thread for bb1, the frame pointer of its consumer(s) is unknown, and it is not even sure there will be one

General Conversion of Control Flow to Data Flow

Problem: in general, the consumer is not known at production time



- Decompose the data dependence bb1→bb4 into bb1→bb2 and bb2→bb4
- Additional dependence: frame pointer of bb2 passed through entry to bb1

Data-Driven Execution

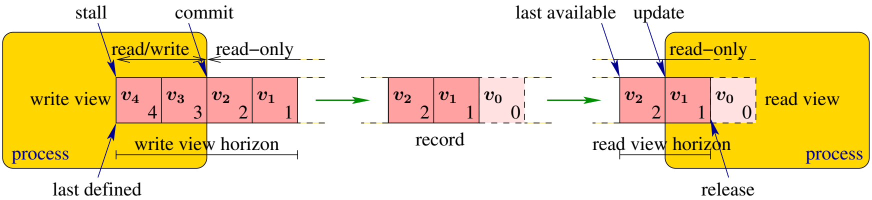
Eliminates with many performance probems!

- Garbage collection, cactus stacks
- Blindness of task scheduler w.r.t. future synchronizations
  - The critical path is hidden
  - Non-urgent tasks waste precious local memory resources
- Memory consumption of suspended, waiting tasks
- Scheduling overhead of task suspension

Remaining challenges

- Allows to implement arbitrary dynamic dependences, but cumbersome: need for "proxy" threads, decoupling thread creation and join from the computational part
- Compilation methods and runtime dependence resolver to let a thread know about its consumers
- Missing a method to aggregate communications across multiple instances of a task

Optimization: Stream Computing

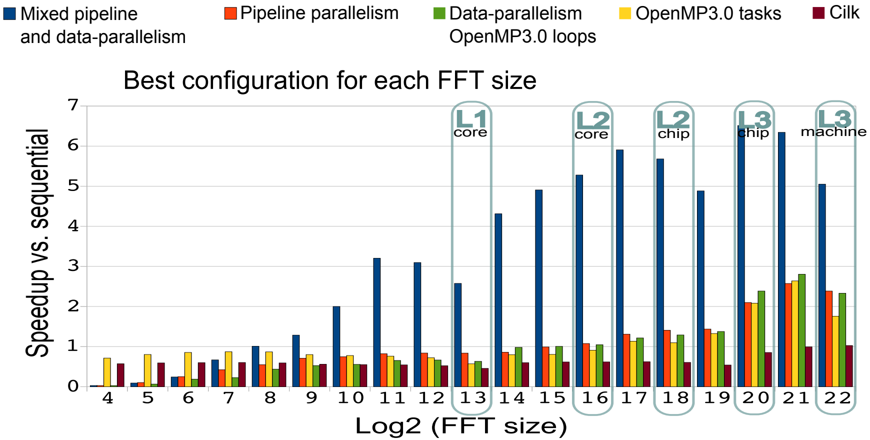


Generalization of Arvind, Nikhil and Pingali's I-structures: index-based MPMC streams

Lightweight runtime

- Lock-free, consensus-free implementation
  - No hardware atomic instruction
  - No memory fence with x86 memory model
- $\approx \mathbf{10}$ cycles per streaming communication cycle
- Enables fine-grain concurrency: very good for local memories

Evaluation on FFT



4-socket Opteron – 16 cores

Data-Flow Stream Computing Based on OpenMP: OpenStream

http://www.di.ens.fr/StreamingOpenMP

A streaming extension of OpenMP (TERAFLUX EU project)

- Dynamic, nested task creation
- First class streams (function arguments/return, heap data structures)
- Unifies streams (w/ sliding windows) and dynamic data flow
- Modular composition (separate compilation)
- Formal semantics: Control-Driven Data Flow (CDDF)
- Prototype in GCC 4.7
- Working on power- and energy-aware scheduling (PHARAON EU project)

Scaling Shared Memory to Manycore Architectures: global address space model

- Data-flow based: inspired by location consistency and DAG consistency
- Region-based cache/publish semantics and relaxed memory model
- Non-partitioned: $\mathcal{V}GAS$: "Virtual" GAS, not $\mathcal{P}GAS$

5. Conclusion and Perspectives

Lessons Learnt

Application programmers

- Do not abandon decades of progress in programming languages, software engineering, and tools
- Unmanaged languages like C, C++ (Fortran?) have a bright future
- Domain-specific languages also

Runtime library and compiler writers

- Define a portable concurrency model for asynchronous tasks
- Scalable and efficient coordination, communication, and synchronization
- Convert portable concurrency into target-specific, in-place computations
- Memory model: formal semantics, and SW support to scale shared memory

Hardware designers

- Invest into a standards-compliant, open source tool chain
- Implement ΦGAS (e.g., P2012) or $\mathcal{V} GAS$ (e.g., MPI clusters, Kalray MPPA)
- Optimize most of the chip for the common automatable case
- Isolate time-predictable area for reactive control applications

Languages and Compilers for Productivity and Efficiency

Albert Cohen

with contributions from Riyadh Baghdadi, Léonard Gérard, Tobias Grosser, Adrien Guatto, Nhat Minh Lê, Feng Li, Antoniu Pop, Jean-Yves Vet, Sven Verdoolaege, and more

INRIA and École Normale Supérieure, Paris
http://www.di.ens.fr/ParkasTeam.html

Workshop on Virtual Machines and Multicore Architectures, September 27–28, 2012

PARKAS Team
Synchronous Kahn Parallelism

- Data-flow synchronous languages, compilers, runtime libraries
- Polyhedral compilation and tools
- Applied to embedded control, parallel programming, compiler construction

Established September 2010, INRIA and École Normale Supérieure, Paris

5 faculty, 4 postdocs, 14 PhD students, 1 engineer

Marc Pouzet

Jean Vuillemin

Albert Cohen

Louis Mandel

Francesco Zappa Nardelli

http://www.di.ens.fr/ParkasTeam.html