Titanium: A High-Performance Java Dialect

Jason Ryder
Matt Beaumont-Gay
Aravind Bappanadu

Titanium Goals

- Design a language that could be used for high performance on some of the most challenging applications
  - Eg. adaptivity in time and space, unpredictable dependencies, data structures that are sparse, hierarchical or pointer-based
- Design a high-level language offering object-orientation with strong typing and safe memory management in the context of high-performance, scalable parallelism

What is Titanium?

- Titanium is an explicitly parallel extension of the Java programming language
  - chosen over the more portable library-based approach because compiler changes would be necessary in either case
- Parallelism achieved through Single-Program Multiple Data (SPMD) and Partitioned Global Address Space (PGAS) models.

Why Titanium Designers Made These Choices for Parallelism

- Decisions to consider when designing a language for parallelism
  1. Will parallelism be expressed explicitly or implicitly?
  2. Is the degree of parallelism static or dynamic?
  3. How do the individual processes interact; data communication and synchronization

- Answers to the first two questions categorize languages into 3 principle categories:
    1. Data-parallel
    2. Task-parallel
    3. Single-program multiple data (SPMD)
- Answers to last question categorize as:
    1. Message passing
    2. Shared memory
    3. Partitioned global address space (PGAS)

Data-Parallel

- Desirable for the semantic simplicity
  - Parallelism determined by the data structures in the program (programmer need not explicitly define parallelism)
  - Parallel operations include element-wise array arithmetic, reduction and scan operations
- Drawbacks
  - Not expressive enough for the most irregular parallel algorithms (e.g. divide-and-conquer parallelism and adaptivity)
  - Relies on a sophisticated compiler and runtime support (less power in the hands of the programmer)

Task-Parallel

- Allows programmer to dynamically create parallelism for arbitrary computations
  - Thereby accommodating expressive parallelization of the most complex of parallel dependencies
- Lacks direct user control over parallel resources
  - Parallelism unfolds at runtime

Single Program Multiple Data

- Static parallelism model
  - A single program executes in each of a fixed number of processes
    - All processes created at program startup and remain until program termination
- Parallelism is *explicit* in the parallel system semantics
- Model offers more flexibility than an implicit model based on data parallelism
- Offers more user-control over performance than either data-parallel or general task-parallel approaches

SPMD cont…

- Processes synchronize with each other at programmer-specified points, otherwise proceed independently
- Most common synchronizing construct is the barrier.
- Also provides locking primitives and synchronous messages

Titanium and SPMD

- Titanium chose SPMD model to place the burden of parallel decomposition explicitly on the programmer
- Provide programmer a transparent model of how the computations would perform on a parallel machine
- Goal is to allow for the expression of the most highly optimized parallel algorithms

Message Passing

- Data movement is explicit
- Allows for coupling communication with synchronization
- Requires a two-sided protocol
- Packing/Unpacking must be done for non-trivial data structures

Shared Memory

- Process can access shared data structure at any time without interrupting other processes
- Shared data structures can be directly represented in memory
- Requires synchronization constructs to control access to shared data (e.g. locks)

Partitioned Global Address Space (PGAS)

- Variation of shared memory model
  - Offers the same semantic model
  - Different performance model
    - The shared memory space is logically *partitioned* between processes
    - Processes have fast access to memory within their own partition
    - Potentially slower access to memory residing in a remote partition
- Typically requires programmer to explicitly state locality properties of all shared data structures

Titanium and PGAS

- The PGAS model can run well on distributed-memory systems, shared-memory multiprocessors and uniprocessors
- The partitioned model provides the ability to start with functional, shared-memory-style code and incrementally tune performance for distributed-memory hardware

Titanium and PGAS cont…

- In Titanium, all objects allocated by a given process will always reside entirely in its own partition of the memory space
- There is an explicit distinction between
  - Shared and private memory
    - Private is typically the processes stack and shared is on the heap
  - local and global pointers
    - performance and static typing benefits

Local vs. Global Pointers

- **Global pointers** may be used to access memory from both the local partition and shared partitions belonging to other processes

- **Local pointers** may only be used to access the process's local partition

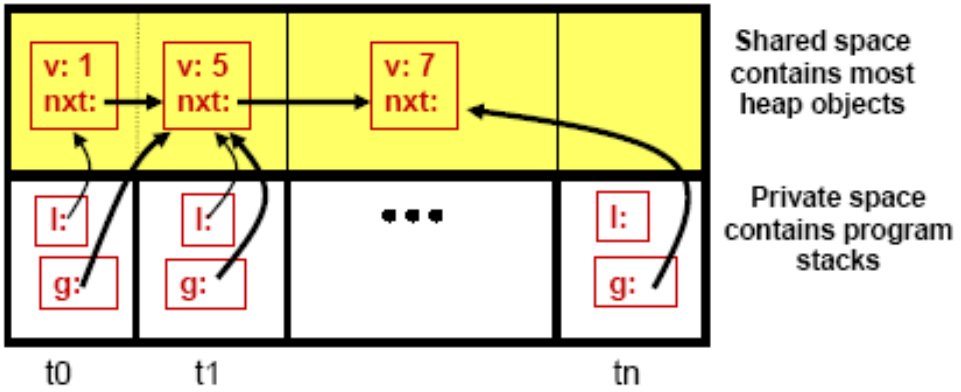

Figure 1: Titanium's Memory Model.

•In Figure 1:

g denotes a global pointer

l denotes a local pointer

nxt is a global pointer

Language Features

- General HPC/scientific computing
- Explicit parallelism

Immutable Classes

- `immutable` keyword in class declaration
- Non-static fields all implicitly `final`
- Cannot be subclass or superclass
- Non-null
- Allows compiler to allocate on stack, pass by value, inline constructor, etc.

Points and Domains

- New built-in types for bounding and indexing N-dimensional arrays
- `Point<`*N*`>` is an N-tuple of integers
- `Domain<`*N*`>` is an arbitrary finite set of `Point<`*N*`>`
  - `RectDomain<`*N*`>` is a rectangular domain
- Can union, intersect, extend, shrink, slice, etc.
- `foreach` loops over the points in a domain in arbitrary order

Grid Types

- Type constructor: `T[`*N*`d]`
- Constructor called with `RectDomain<`*N*`>`
- Indexed with `Point<N>`
- `overlap` keyword in method declaration allows specified grid-typed formals to alias each other

Memory-Related Type Qualifiers

- Variables are global unless declared `local` (to statically eliminate communication check)
- Variables of reference types are shared unless declared `nonshared`
  - May also be `polyshared`

I/O and Data Copying

- Efficient bulk I/O on arrays
- Explicit gather/scatter for copying sparse arrays
- Non-blocking array copying

Maintaining Global Synchronization

- Some expressions are *single-valued*, e.g.:
  - Constants
  - Variables or parameters declared as `single`
  - `e1 + e2` if `e1` and `e2` are single-valued
- Some classes of statements have *global effects*, e.g:
  - Assignment to `single` variables
  - `broadcast`

Maintaining Global Synchronization

- "An `if` statement whose condition is not single-valued cannot have statements with global effects as its branches."
- In `e.m(...)`, if `m` may be `m0` with global effects, `e` must be single-valued
- Etc., etc.

Barriers

- `Ti.barrier()` causes a process to wait until all other processes have reached *the same textual instance* of the barrier
- "Barrier inference" technique used to detect possible deadlocks at compile time

`broadcast`

- `broadcast e from p`
- `p` must be single-valued
- All processes but `p` wait at the expression
- `e` is evaluated on `p`
- The value is returned in all processes

`exchange`

- `A.exchange(e)`
- Domain of `A` must be superset of the domain of process IDs
- Provides an implicit barrier
- In all processes, `A[i]` gets process *i*'s value of `e`

Demo!

References

- Alexander Aiken and David Gay. "Barrier Inference." *Proc. POPL,* 2005.
- P. N. Hilfinger (ed.), Dan Bonachea, et al. "Titanium Language Reference Manual." UC Berkeley EECS Technical Report UCB/EECS-2005-15.1, August 2006.
- Katherine Yelick, Paul Hilfinger, et al. "Parallel Languages and Compilers: Perspective from the Titanium Experience." *International Journal of High Performance Computing Applications,* Vol. 21, No. 3, 266-290, 2007.