Titanium Performance and Potential: an NPB Experimental Study

Kaushik Datta, Dan Bonachea, and Katherine Yelick

http://titanium.cs.berkeley.edu

LCPC 2005

U.C. Berkeley

October 20, 2005

Take-Home Messages

- **Titanium:**
  - allows for elegant and concise programs
  - gets comparable performance to Fortran+MPI on three common yet diverse scientific kernels (NPB)
  - is well-suited to real-world applications
  - is portable (runs everywhere)

NAS Parallel Benchmarks

- **Conjugate Gradient (CG)**
  - *Computation*: Mostly sparse matrix-vector multiply (SpMV)
  - *Communication*: Mostly vector and scalar reductions
- **3D Fourier Transform (FT)**
  - *Computation*: 1D FFTs (using FFTW 2.1.5)
  - *Communication*: All-to-all transpose
- **Multigrid (MG)**
  - *Computation*: 3D stencil calculations
  - *Communication*: Ghost cell updates

Titanium Overview

- **Titanium is a Java dialect for parallel scientific computing**
  - No JVM, no JIT, and no dynamic class loading
- **Titanium is extremely portable**
  - Ti compiler is source-to-source, and first compiles to C for portability
  - Ti programs run everywhere- uniprocessors, shared memory, and distributed memory systems
- **All communication is one-sided for performance**
  - GASNet communication system (not MPI)

Presented Titanium Features

- **Features in addition to standard Java:**
  - Flexible and efficient multi-dimensional arrays
  - Built-in support for multi-dimensional domain calculus
  - Partitioned Global Address Space (PGAS) memory model
  - Locality and sharing reference qualifiers
  - Explicitly unordered loop iteration
  - User-defined immutable classes
  - Operator-overloading
  - Efficient cross-language support
- **Many others not covered…**

Titanium Arrays

- **Ti Arrays are created and indexed using *points*:**

```
double [3d] gridA = new double [[-1,-1,-1]:[256,256,256]];
```

(MG)

Lower Bound — Upper Bound

- **`gridA` has a rectangular index set (*RectDomain*) of all points in box with corners `[-1,-1,-1]` and `[256,256,256]`**
- ***Points* and *RectDomains* are first-class types**
- **The power of Titanium arrays lies in:**
  - Generality: indices can start at any point
  - Views: one array can be a subarray of another

Foreach Loops

- ***Foreach*** **loops allow for unordered iterations through a** ***RectDomain*****:**

```
public void square(double [3d] gridA, double [3d] gridB) {
   foreach (p in gridA.domain()) {
    gridB[p] = gridA[p] * gridA[p];
   }
}
```

- **These loops:**
  - allow the compiler to reorder execution to maximize performance
  - require only one loop even for multidimensional arrays
  - avoid off-by-one errors common in *for* loops

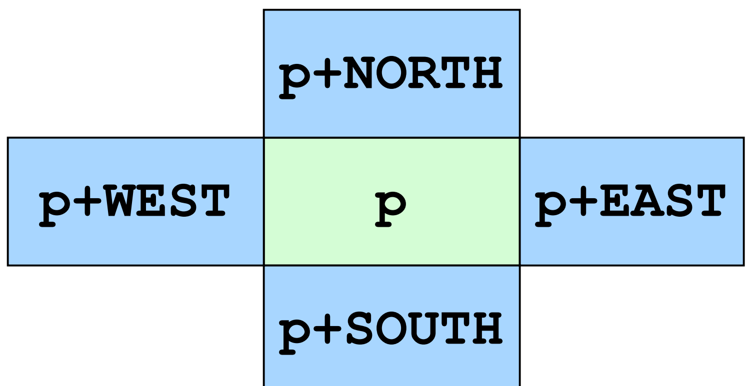
Point Operations

- **Titanium allows for arithmetic operations on Points:**

```
final Point<2> NORTH = [0,1], SOUTH = [0,-1],
               EAST = [1,0],  WEST = [-1,0];

foreach (p in gridA.domain()) {
  gridB[p] = S0 * gridA[p] +
             S1 * ( gridA[p + NORTH] + gridA[p + SOUTH] +
                    gridA[p + EAST] +  gridA[p + WEST] );
         }
```

- **This makes the MG stencil code more readable and concise**

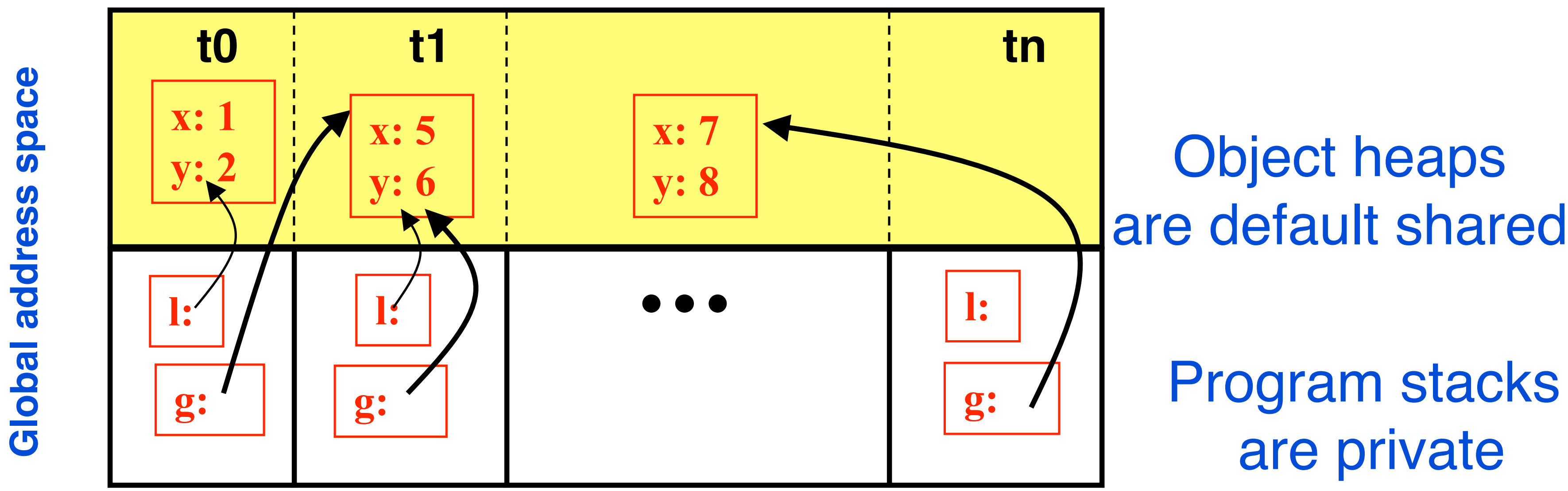


Titanium Parallelism Model

- **Ti uses an SPMD model of parallelism**
  - Number of threads is fixed at program startup
  - Barriers, broadcast, reductions, etc. are supported
- **Programmability using a Partitioned Global Address Space (i.e., direct reads and writes)**
  - Programs are portable across shared/distributed memory
  - Compiler/runtime generates communication as needed
  - User controls data layout locality; key to performance

PGAS Memory Model

- **Global address space is logically partitioned**
  - Independent of underlying hardware (shared/distributed)
  - Data structures can be spread over partitions of shared space
- **References (pointers) are either local or global (meaning possibly remote)**



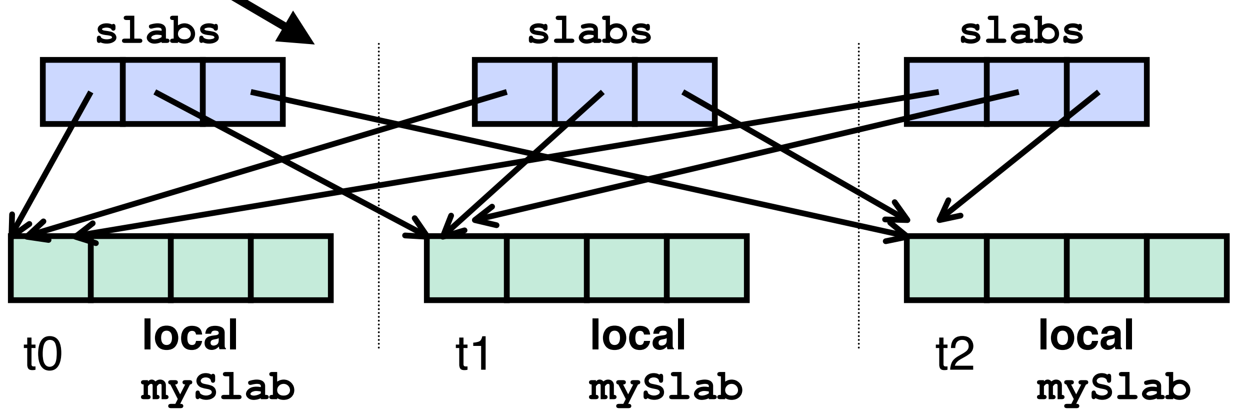
Distributed Arrays

- **Titanium allows construction of distributed arrays in the shared Global Address Space:**

```
double [3d] mySlab = new double [startCell:endCell];

// "slabs" array is pointer-based directory over all procs
double [1d] single [3d] slabs = new double [0:Ti.numProcs()-1]
   single [3d];
slabs.exchange(mySlab);
```

(FT)



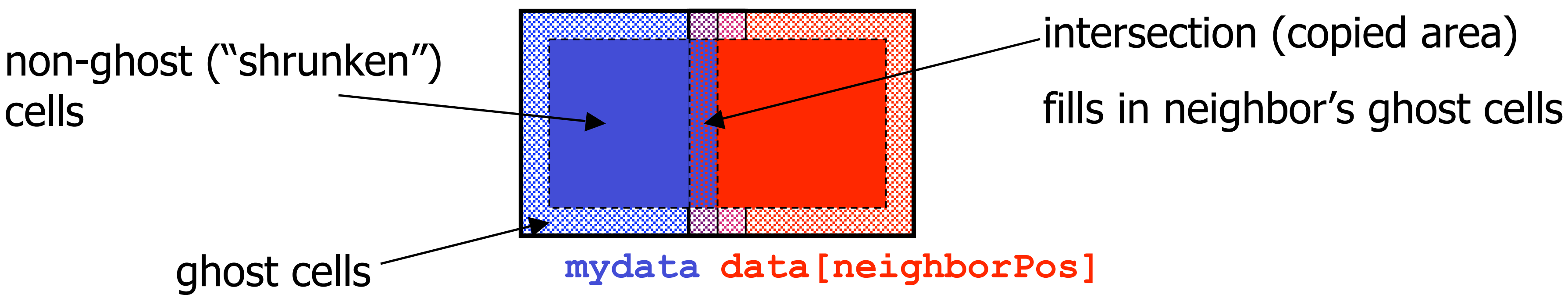
Domain Calculus and Array Copy

- **Full power of Titanium arrays combined with PGAS model**
- **Titanium allows set operations on RectDomains:**

```
// update overlapping ghost cells of neighboring block
data[neighborPos].copy(myData.shrink(1));
```

(MG)

- **The copy is only done on intersection of array RectDomains**
- **Titanium also supports nonblocking array copy**



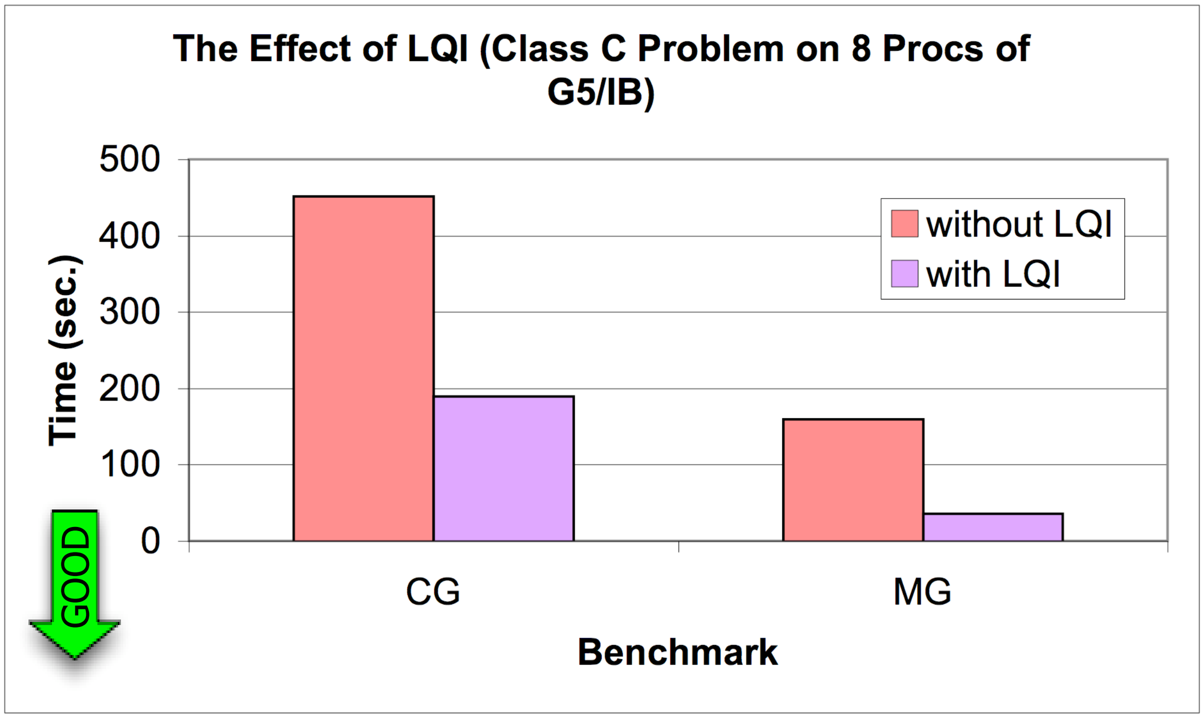
The Local Keyword and Compiler Optimizations

- ***Local*** **keyword ensures that compiler statically knows that data is local:**

```
double [3d] myData = (double [3d] local) data[myBlockPos];
```

- **This allows the compiler to use more efficient native pointers to reference the array**
  - Avoid runtime check for local/remote
  - Use more compact pointer representation
- **Titanium optimizer can often automatically propagate locality info using *Local Qualifier Inference* (LQI)**

Is LQI (Local Qualifier Inference) Useful?



- **LQI does a solid job of propagating locality information**
- **Speedups:**
  - CG- 58% improvement
  - MG- 77% improvement

Immutable Classes

- **For small objects, would sometimes prefer:**
  - to avoid level of indirection and allocation overhead
  - to pass by value (copying of entire object)
  - especially when immutable (fields never modified)
    - Extends idea of primitives to user-defined data types
- **Example: Complex number class**

```
immutable class Complex {
  // Complex class is now unboxed
  public double real, imag;
  …
}
```

(FT)

No assignment to fields
outside of constructors

Operator Overloading

- **For convenience, Titanium allows operator overloading**
  - Overloading in Complex makes the FT benchmark more readable
  - Similar to operator overloading in C++

```
immutable class Complex {
  public double real;
  public double imag;
  public Complex op+(Complex c) {
    return new Complex(c.real + real, c.imag + imag);
  }
}
Complex c1 = new Complex(7.1, 4.3);
Complex c2 = new Complex(5.4, 3.9);
Complex c3 = c1 + c2;
```

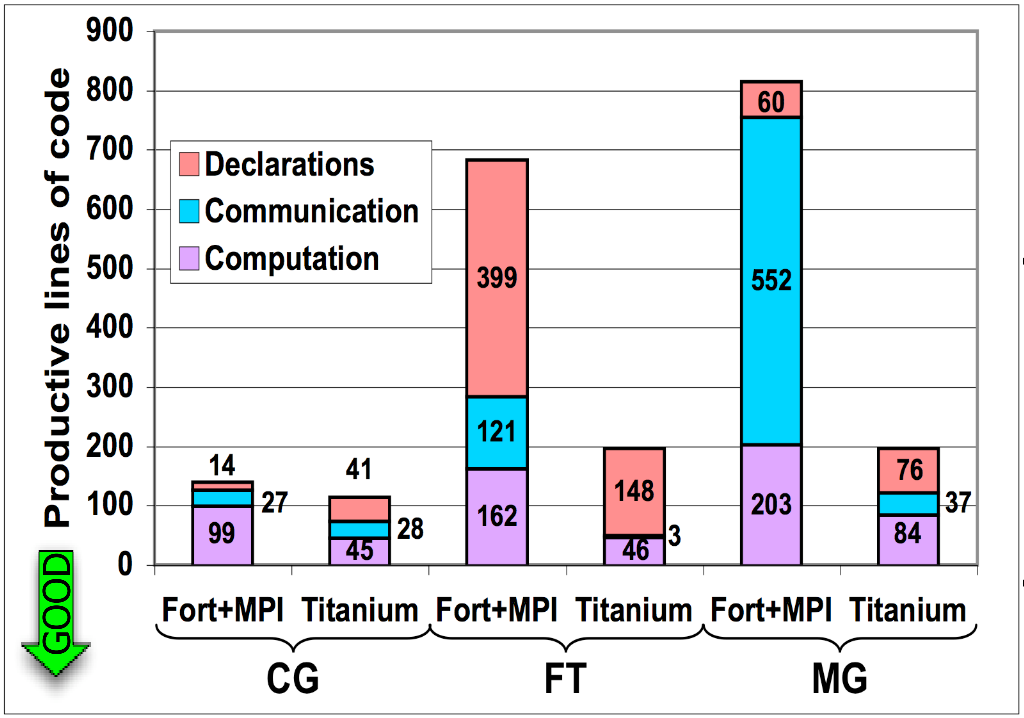
(FT)

"+" is overloaded to add Complex objects

Cross-Language Calls

- **Titanium supports efficient calls to kernels/libraries in other languages**
  - no data copying required
- **Example: the FT benchmark calls the FFTW library to perform the local 1D FFTs**
- **This encourages:**
  - shorter, cleaner, and more modular code
  - the use of tested, highly-tuned libraries

Are these features expressive?



- **Compared line counts of timed, uncommented portion of each program**
- **MG and FT disparities mostly due to Ti domain calculus and array copy**
- **CG line counts are similar since Fortran version is already compact**

Testing Platforms

- **Opteron/InfiniBand (NERSC / Jacquard):**
  - *Processor*: Dual 2.2 GHz Opteron (320 nodes, 4 GB/node)
  - *Network*: Mellanox Cougar InfiniBand 4x HCA
- **G5/InfiniBand (Virginia Tech / System X):**
  - *Processor*: Dual 2.3 GHz G5 (1100 nodes, 4 GB/node)
  - *Network*: Mellanox Cougar InfiniBand 4x HCA

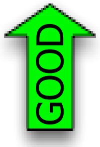
Problem Classes

	Matrix or Grid Dimensions	Iterations
CG Class C	$150{,}000^2$	75
CG Class D	$1{,}500{,}000^2$	100
FT Class C	512^3	20
MG Class C	512^3	20
MG Class D	1024^3	50

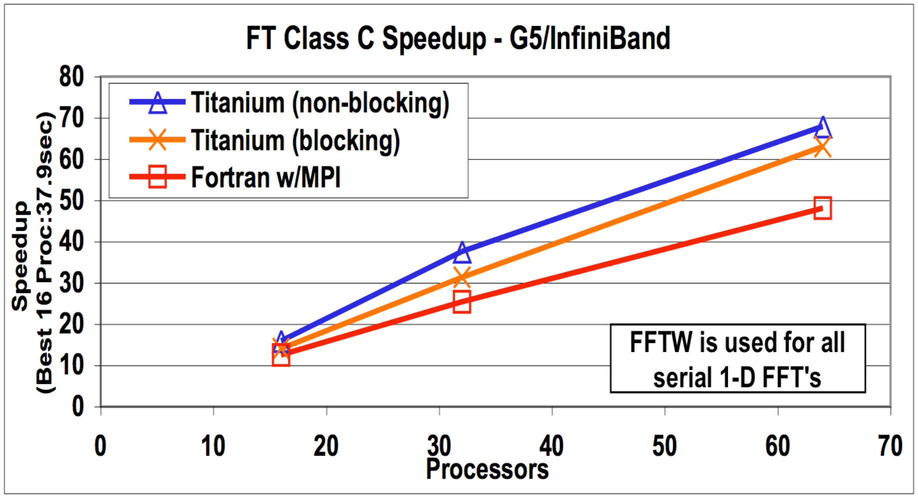
All problem sizes shown are relatively large

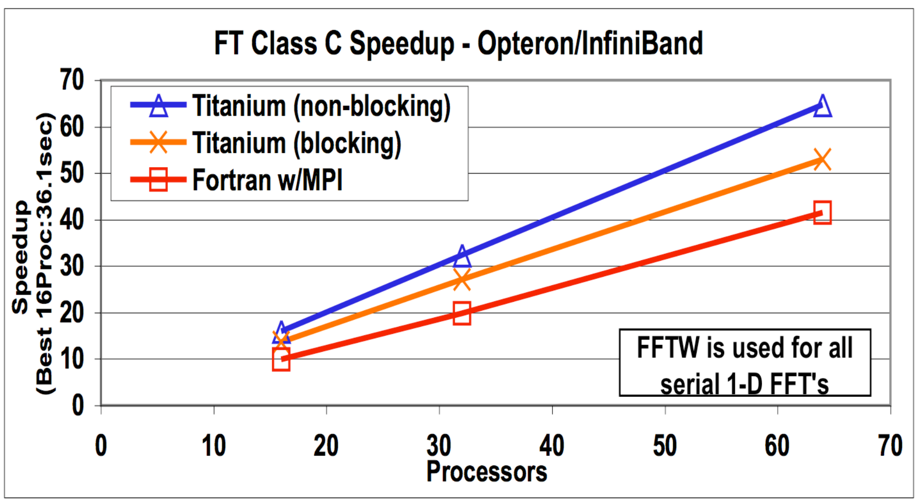
Data Collection and Reporting

- **Each data point was run three times, and the minimum of the three is reported**
- **For a given number of procs, the Fortran and Titanium codes were run on the same nodes (for fairness)**
- **All the following speedup graphs use the *best time* at the lowest number of processors as the baseline for the speedup**



FT Speedup



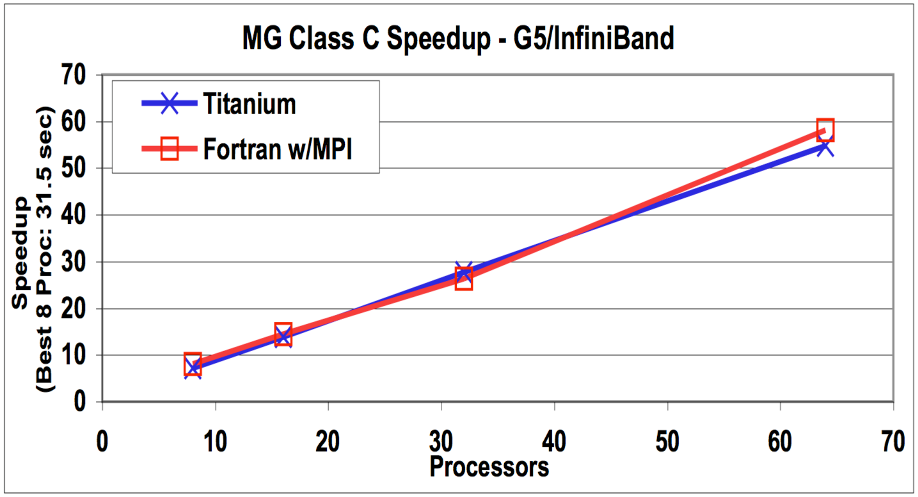


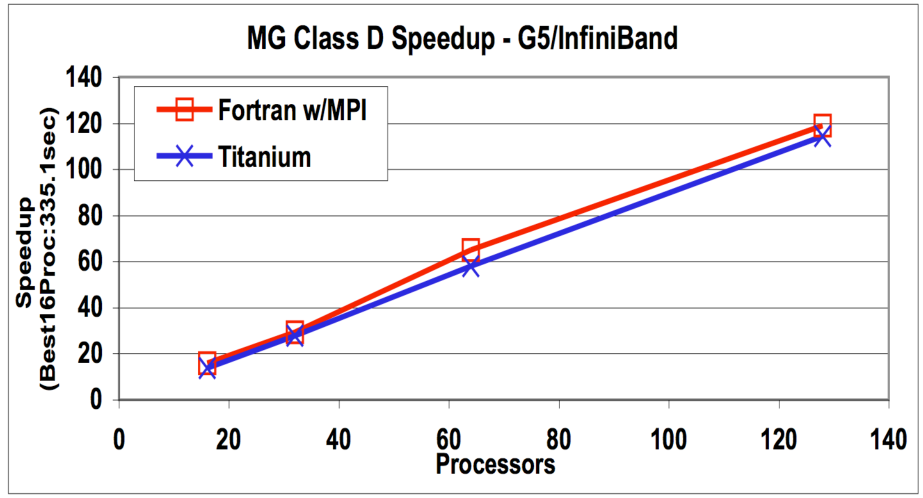
- **All versions of the code use FFTW 2.1.5 for the serial 1D FFTs**
- **Nonblocking array copy allows for comp/comm overlap**
- **Max Mflops/proc:**

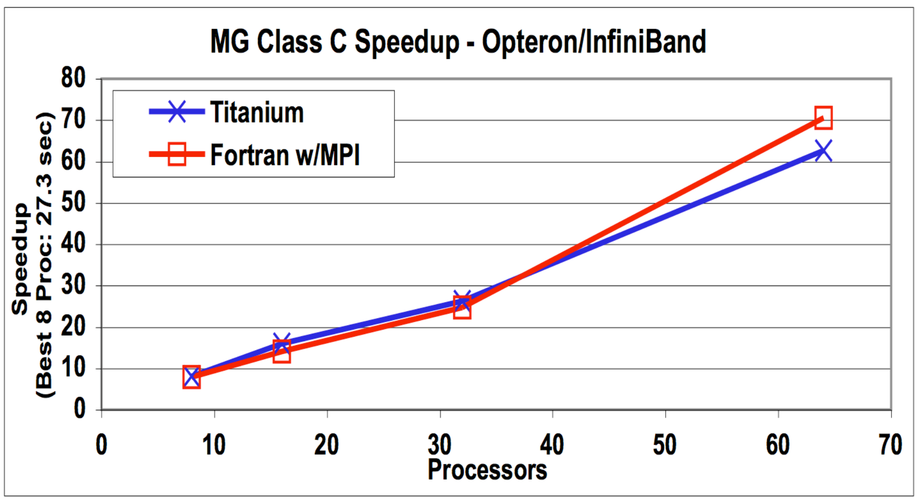
	For	Ti(bl)	Ti(nbl)
G5	238	294	350
Opt.	203	268	318

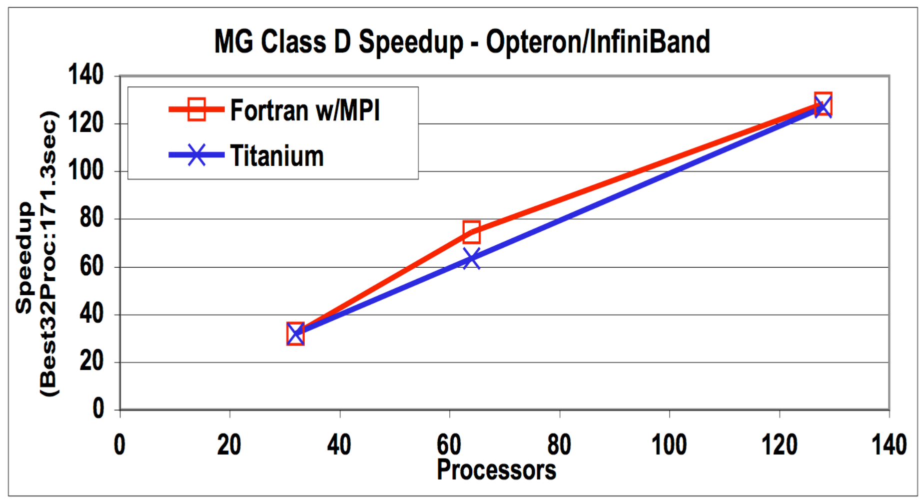

GOOD

MG Speedup


MG Class C Speedup - G5/InfiniBand
Titanium
Fortran w/MPI
Speedup
(Best 8 Proc: 31.5 sec)
Processors
0
10
20
30
40
50
60
70

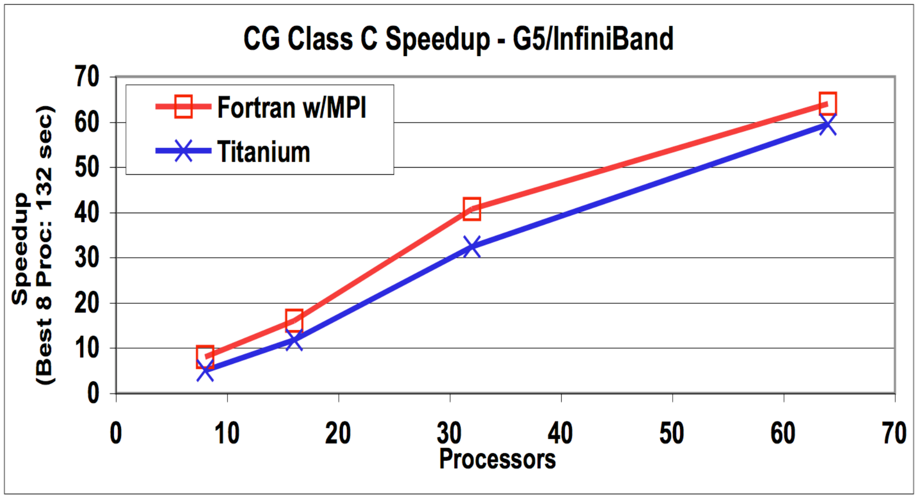

MG Class D Speedup - G5/InfiniBand
Fortran w/MPI
Titanium
Speedup
(Best16Proc:335.1sec)
Processors
0
20
40
60
80
100
120
140

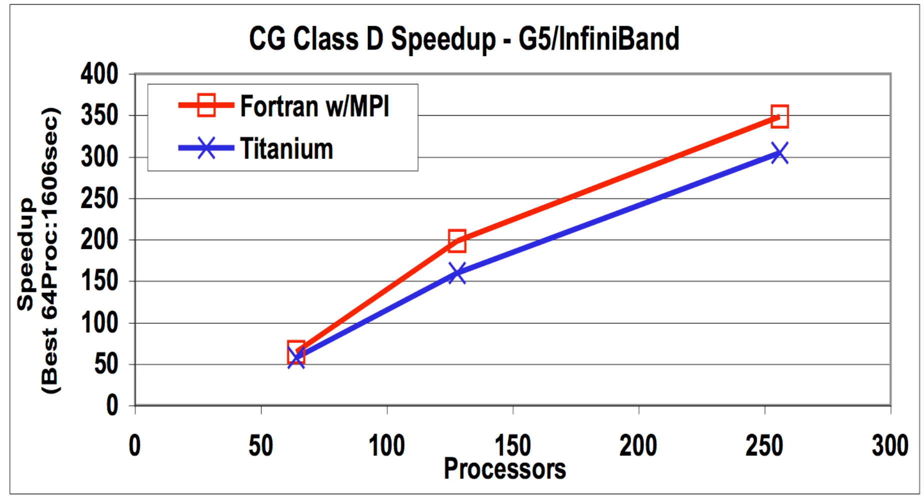

MG Class C Speedup - Opteron/InfiniBand
Titanium
Fortran w/MPI
Speedup
(Best 8 Proc: 27.3 sec)
Processors
0
10
20
30
40
50
60
70
80

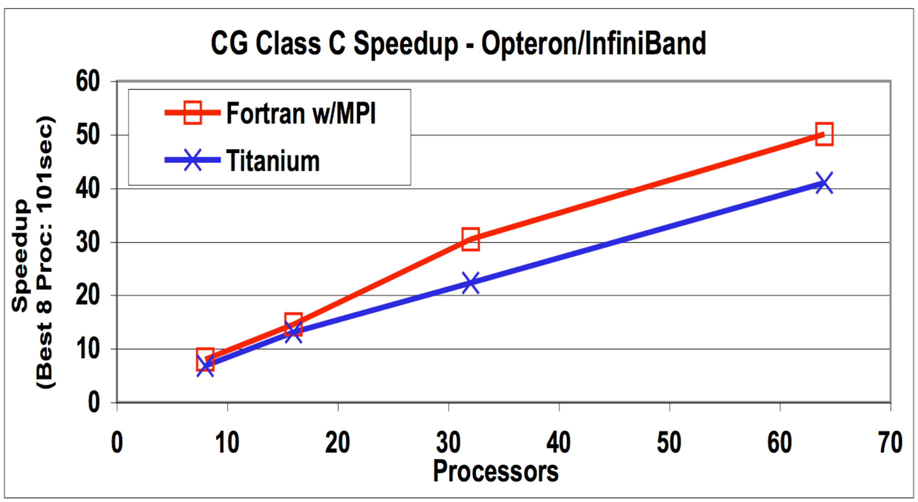

MG Class D Speedup - Opteron/InfiniBand
Fortran w/MPI
Titanium
Speedup
(Best32Proc:171.3sec)
Processors
0
20
40
60
80
100
120
140

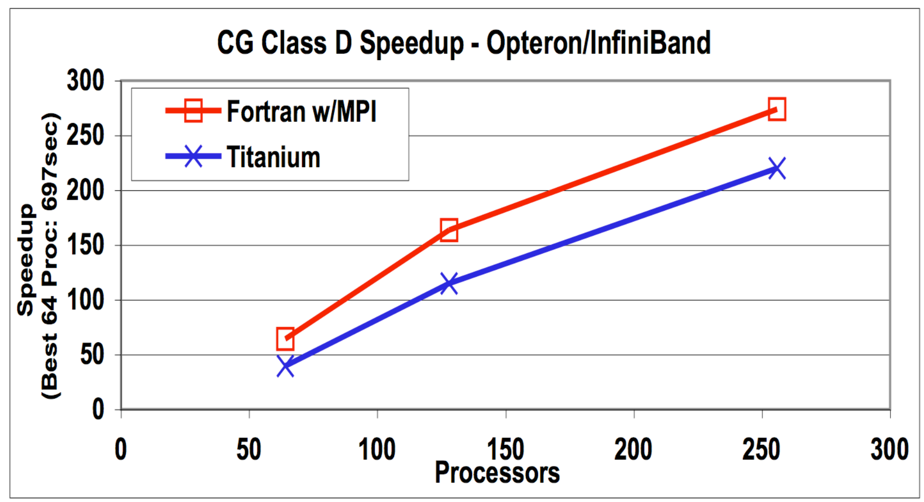

GOOD

CG Speedup


CG Class C Speedup - G5/InfiniBand
Speedup
(Best 8 Proc: 132 sec)
Processors
Fortran w/MPI
Titanium
0
10
20
30
40
50
60
70


CG Class D Speedup - G5/InfiniBand
Speedup
(Best 64Proc:1606sec)
Processors
Fortran w/MPI
Titanium
0
50
100
150
200
250
300
350
400


CG Class C Speedup - Opteron/InfiniBand
Speedup
(Best 8 Proc: 101sec)
Processors
Fortran w/MPI
Titanium
0
10
20
30
40
50
60
70


CG Class D Speedup - Opteron/InfiniBand
Speedup
(Best 64 Proc: 697sec)
Processors
Fortran w/MPI
Titanium
0
50
100
150
200
250
300

Other Applications in Titanium

- **Larger Applications**
  - Heart and cochlea simulations (E. Givelberg, K. Yelick, A. Solar-Lezama, J. Su)
  - AMR Elliptic PDE solver (P. Colella, T. Wen)
- **Other Benchmarks and Kernels**
  - Scalable Poisson solver for infinite domains
  - Unstructured mesh kernel: EM3D
  - Dense linear algebra: LU, MatMul
  - Tree-structured n-body code
  - Finite element benchmark

Conclusions

- **Titanium:**
  - Captures many abstractions needed for common scientific kernels
  - Allows for more productivity due to fewer lines of code
  - Performs comparably and sometimes better to Fortran w/MPI
  - Provides more general distributed data layouts and irregular parallelism patterns for real-world problems (e.g., heart simulation, AMR)