Communication Optimizations in Titanium Programs

Jimmy Su

Study Communication Optimization

- **Benchmarks**
  - —Gups
  - —Sparse Matvec
  - —Jacobi
  - —Particle in Cell
- **Machines used in experiments**
  - —Seaborg (IBM SP)
  - —Millennium

Hand Optimizations

- **Prefetching (moving reads up)**
- **Moving syncs down**
- **C code generated by the Titanium compiler is modified manually to do the above optimizations**

Characteristics of the Benchmarks

- **Source code was not optimized**
- **There are more remote reads than remote writes**
- **Source code uses small messages instead of pack/unpack**

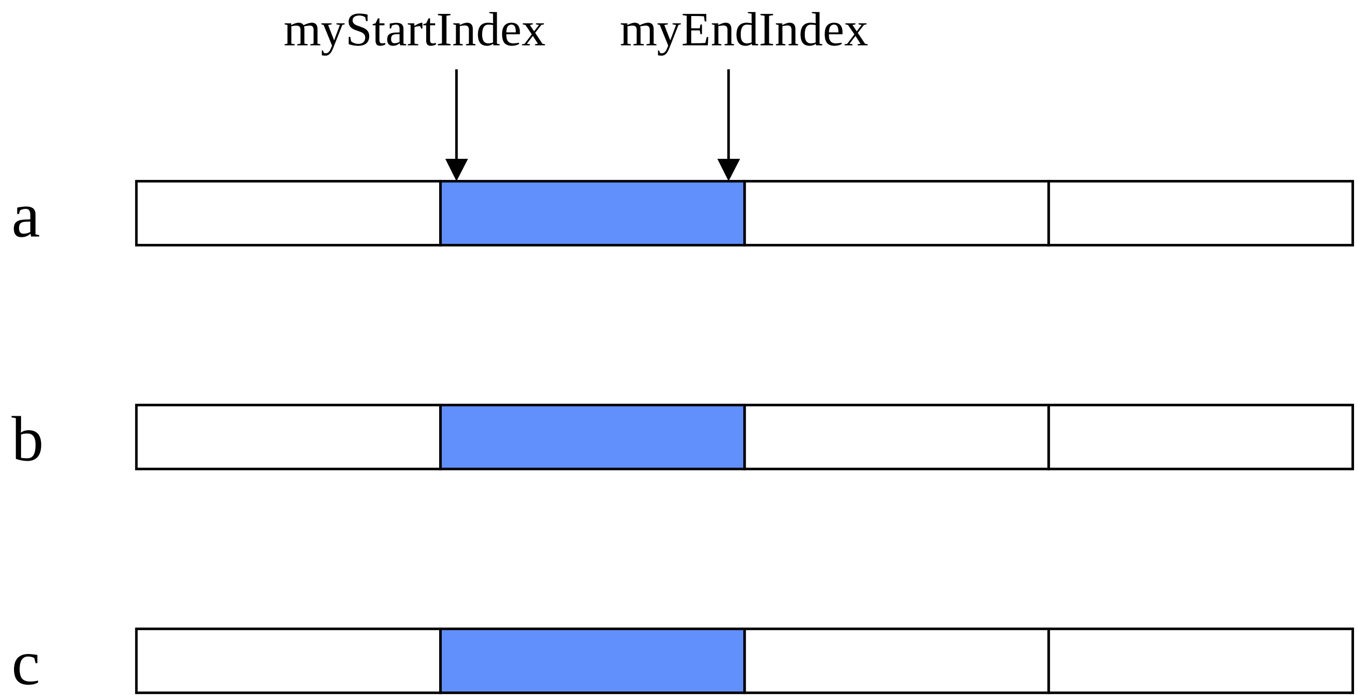
Observations

- Pros
  - —Hand optimization does pay off
    - Gups 14% speed up
    - Jacobi 5% speed up
    - Sparse Matvec 45% speed up
- Cons
  - —The optimizations can only be done automatically on regular problems
    - Alias analysis too conservative
  - —Alternative solution for regular problems uses array copy
    - Titanium has highly optimized array copy routines

Inspector Executor

- Developed by Joel Saltz and others at University of Maryland in the early 90's
- Goal is to hide latency for problems with irregular accesses
- A loop is compiled into two phases, an inspector and an executor
  - —The inspector examines the data access pattern in the loop body and creates a schedule for fetching the remote values
  - —The executor retrieves remote values according to the schedule and executes the loop
- A schedule may be reused if the access pattern is the same for multiple iterations

Inspector Executor Example



Inspector Executor Pseudo Code

```
for iteration = 1 to n
    for i = myStartIndex to myEndIndex
        a[i] = b[i] + c[ia[i]]
    end
    c.copy(a)
end
```

→

```
//inspector phase
for i = myStartIndex to myEndIndex
    a[i] = b[i] + c.inspect(ia[i])
end

//create the communication schedule
c.schedule()

for iteration = 1 to n
    //fetch the remote values according to the
    //communication schedule
    c.fetch()
    for i = myStartIndex to myEndIndex
        a[i] = b[i] + c.execute(ia[i])
    end
    c.copy(a)
end
```

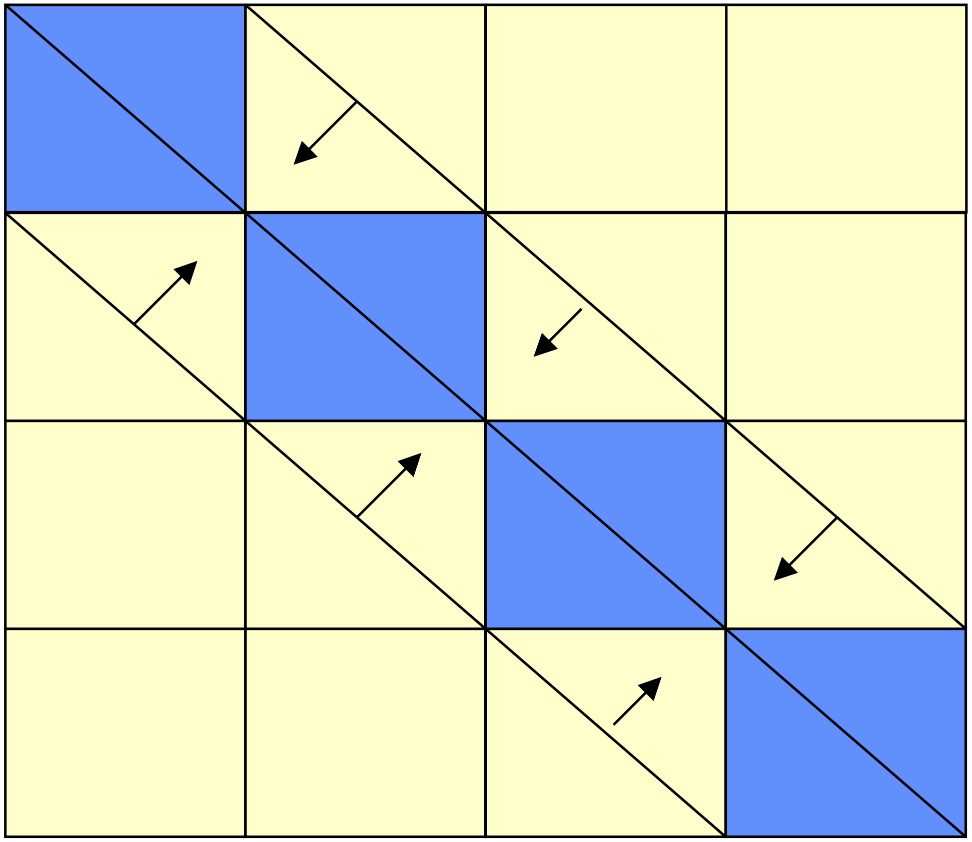
Roadmap

- Introduced distributed array type
- First implemented by hand
- Currently working on a prototype in the compiler

Conjugate Gradient

- 4096x4096 matrices
- 0.07% of matrix entries are non-zeros
- Varies the percent of non-local accesses from 0% to 64%
- 8 processors on 2 nodes with 4 processors on each node
- Only the sparse matvec part is modified to use inspector executor
- The running time of 500 iterations was measured
- Seaborg (IBM SP)

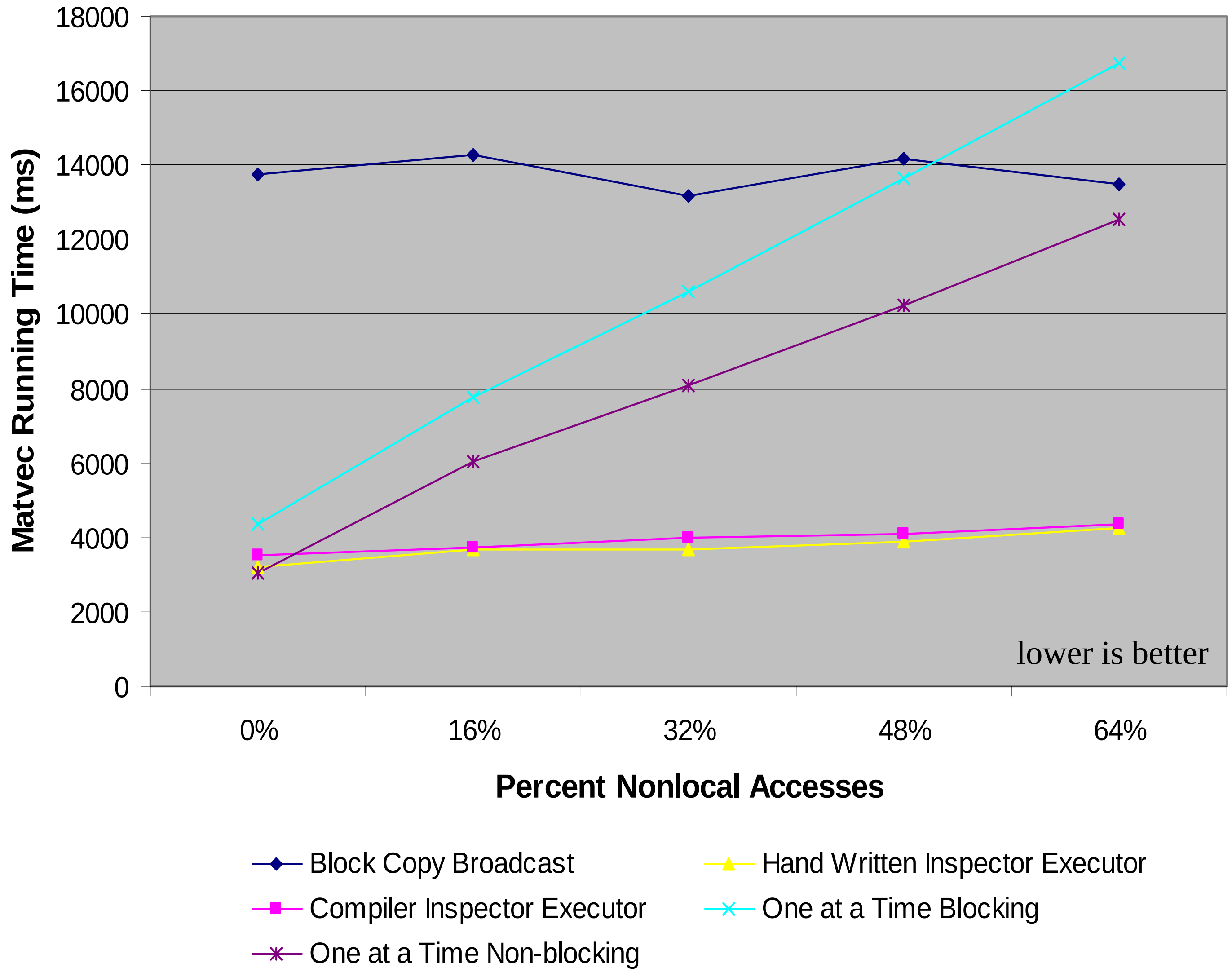
Synthetic Matrices For Benchmark



Description of the Benchmark

- Compiler generated
  - Block copy broadcast
  - Compiler inspector executor
  - One at a time blocking
- Hand edited
  - Hand written inspector executor
  - One at a time non-blocking

Sparse Matvec

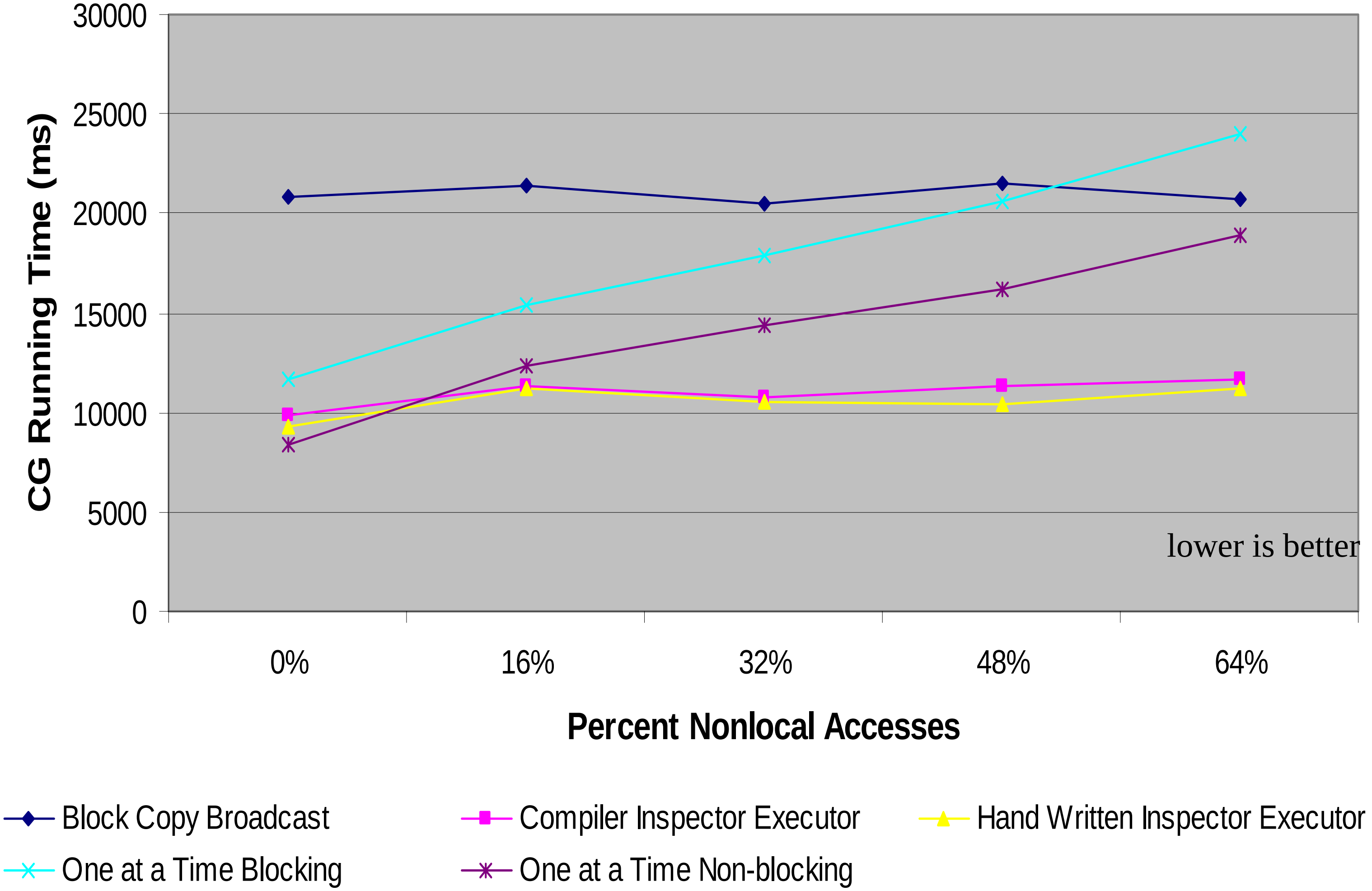


Problem size:

4096x4096 matrix

0.07% fill rate

Full Conjugate Gradient



Problem size:

4096x4096 matrix

0.07% fill rate

Future Work

- Analysis on when the inspector executor transformation is legal
- Investigate the uniprocessor performance of sparse matvec
- Apply inspector executor in UPC
- Run benchmark on matrices with different structures
- Automatically finding a location to place the communication code
- More benchmarks that utilize inspector executor
- Alternative scheduling strategies