Multiparty Computation made Practical

Using the Virtual Ideal Functionality Framework

Martin Geisler
⟨mg@daimi.au.dk⟩

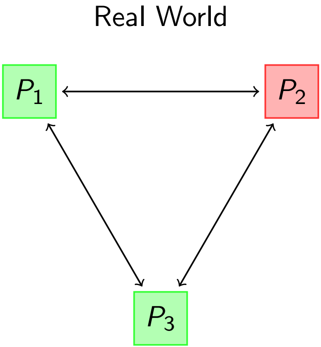
BRICS
Department of Computer Science
University of Aarhus
Denmark

ECRYPT Research Meeting
June 23–25, 2008

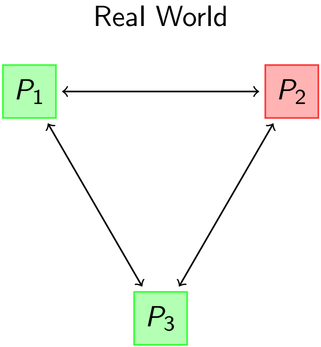
Outline

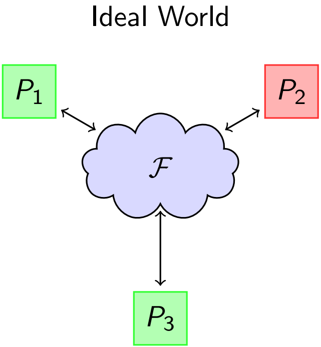
- Background
  - Setting
  - Related Projects
- Design Goals
  - Automatic Parallel Execution
  - Program Counters
- Benchmark Results
  - Multiplications
  - Comparisons
- Possible Improvements
  - Program Counters
  - Memory Overheads
  - Debugging
- Conclusion

Setting

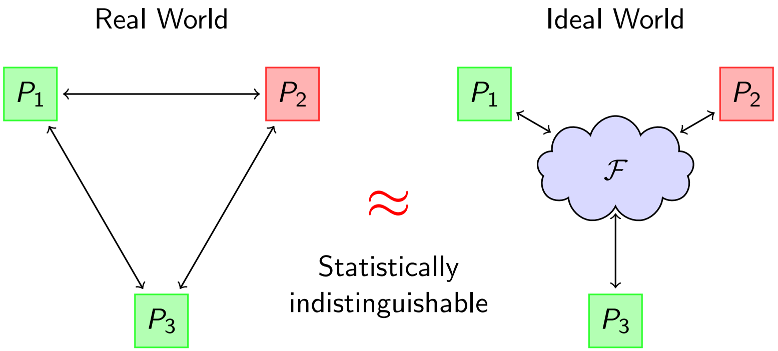


Setting





Setting



Real World

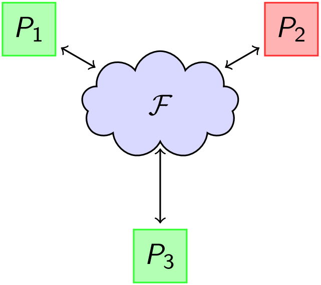
Ideal World

Statistically indistinguishable

Functionality provided by VIFF

- Input
  - Shamir secret sharing

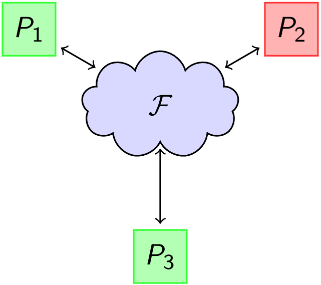
Ideal World



Functionality provided by VIFF

- Input
  - Shamir secret sharing
- Output
  - Reconstruct shares

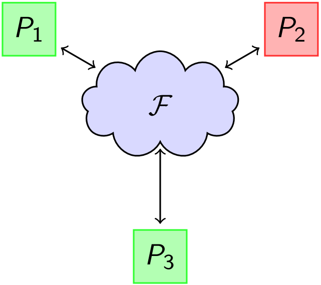
Ideal World



Functionality provided by VIFF

- Input
  - Shamir secret sharing
- Output
  - Reconstruct shares
- Addition
  - Local addition
  - No communication

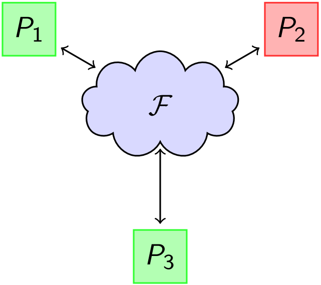
Ideal World



Functionality provided by VIFF

- Input
  - Shamir secret sharing
- Output
  - Reconstruct shares
- Addition
  - Local addition
  - No communication
- Multiplication
  - Local multiplication
  - Resharing

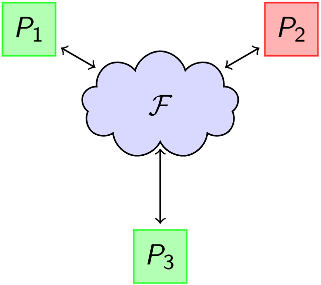
Ideal World



Functionality provided by VIFF

- Input
  - Shamir secret sharing
- Output
  - Reconstruct shares
- Addition
  - Local addition
  - No communication
- Multiplication
  - Local multiplication
  - Resharing
- Comparison
  - Two protocols

Ideal World

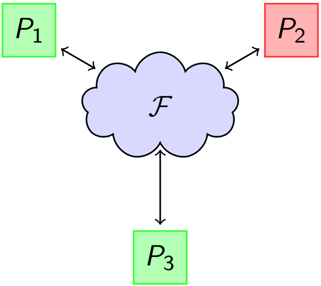


Functionality provided by VIFF

- Input
  - Shamir secret sharing
- Output
  - Reconstruct shares
- Addition
  - Local addition
  - No communication
- Multiplication
  - Local multiplication
  - Resharing
- Comparison
  - Two protocols

All over arbitrary finite fields.

Ideal World



Related Projects

- FairPlay project (Haifa, Israel)
    - Yao-garbled circuits for 2 players
    - Java implementation
    - Working on multiparty computation

Related Projects

- FairPlay project (Haifa, Israel)
  - Yao-garbled circuits for 2 players
  - Java implementation
  - Working on multiparty computation
- SCET project (Aarhus, Denmark)
  - General multiparty computations
  - C# implementation

Related Projects

- FairPlay project (Haifa, Israel)
  - Yao-garbled circuits for 2 players
  - Java implementation
  - Working on multiparty computation
- SCET project (Aarhus, Denmark)
  - General multiparty computations
  - C# implementation
- SIMAP project (Aarhus, Denmark)
  - General multiparty computations
  - Java implementation
  - Some work done on a domain specific language

Outline

- Background
  - Setting
  - Related Projects
- Design Goals
  - Automatic Parallel Execution
  - Program Counters
- Benchmark Results
  - Multiplications
  - Comparisons
- Possible Improvements
  - Program Counters
  - Memory Overheads
  - Debugging
- Conclusion

VIFF Goals

- Easy to use for the programmer:

```
x = 2 * a - b * c
```

VIFF Goals

- Easy to use for the programmer:

```
x = 2 * a − b * c
```

- Automatically run things in parallel:

```
x = a * b
y = c * d
z = e * f
```

VIFF Goals

- Easy to use for the programmer:

```
x = 2 * a − b * c
```

- Automatically run things in parallel:

```
x = a * b
y = c * d
z = e * f
```

- Extensible with new operations:

```
def max(a, b):
    c = a > b
    return c * a + (1 − c) * b
```

Parallel Execution

- Networks have significant latency
- Want to run many operations in parallel
- Including primitive and compound operations

Example: Hamming Distance

```
def xor(a, b):
    assert a.field is b.field
    if a.field is GF256:
        return a + b
    else:
        return a + b - 2 * a * b
```

- Straight-forward exclusive-or
- Fast for $GF(2^8)$ elements
- Slower for $\mathbb{Z}_p$ elements
- (Already part of VIFF)

Example: Hamming Distance

```
def xor(a, b):
    assert a.field is b.field
    if a.field is GF256:
        return a + b
    else:
        return a + b − 2 ∗ a ∗ b
```

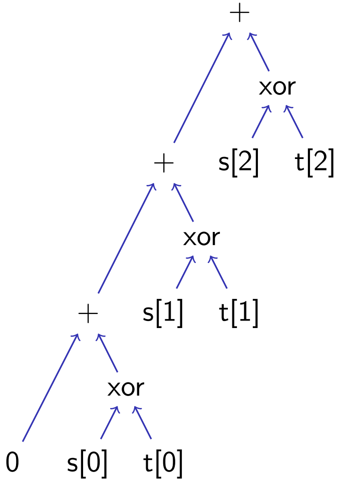
- Straight-forward exclusive-or
- Fast for $GF(2^8)$ elements
- Slower for $\mathbb{Z}_p$ elements
- (Already part of VIFF)

```
def hamming(s, t):
    assert len(s) == len(t)
    distance = 0
    for i in range(len(s)):
        distance += xor(s[i], t[i])
    return distance
```

- Hamming distance
- xor calls should run in parallel

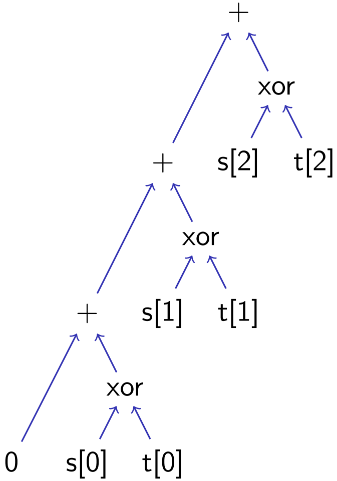
Hamming Distance Execution Tree

hamming(s, t) translates to:

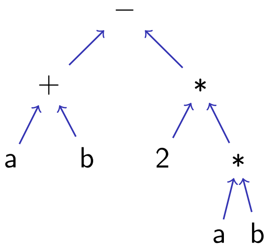


Hamming Distance Execution Tree

hamming(s, t) translates to:



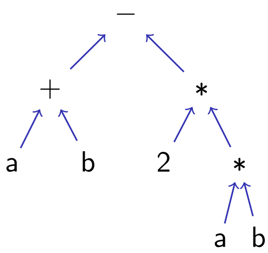
xor(a, b) translates to:



VIFF Execution Strategy

- Create execution tree as we go along
- Destroy execution tree from bottom up
  - Each node waits on nodes below
  - Bottom nodes wait on network traffic
- Composable: just plug new operations into tree!

Program Counters



- No parsing — execution tree never fully constructed
- No fixed evaluation order
- But we must identify results

Program Counters Implementation

- First attempt:
  - Each player increments a global counter
  - Fails because of asynchronous execution

Program Counters Implementation

- First attempt:
  - Each player increments a global counter
  - Fails because of asynchronous execution
- Working solution:
  - Manually “weave” a program counter through program
  - Tedious, easy to forget to increment program counter

Program Counters Implementation

- First attempt:
  - Each player increments a global counter
  - Fails because of asynchronous execution
- Working solution:
  - Manually “weave” a program counter through program
  - Tedious, easy to forget to increment program counter
- Current solution:
  - Runtime methods wrapped by a decorator
  - Calculated automatically based on call stack

Outline

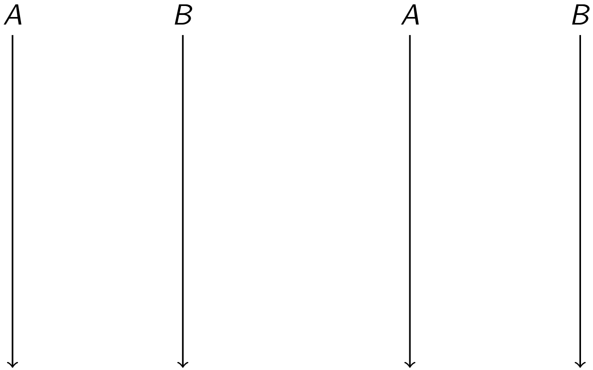
- Background
  - Setting
  - Related Projects
- Design Goals
  - Automatic Parallel Execution
  - Program Counters
- Benchmark Results
  - Multiplications
  - Comparisons
- Possible Improvements
  - Program Counters
  - Memory Overheads
  - Debugging
- Conclusion

Performance Results

- Tested on 3 machines: USA, Norway, and Denmark
- Results for VIFF 0.4 (VIFF 0.6 is similar or better)
- Tested multiplications and comparisons

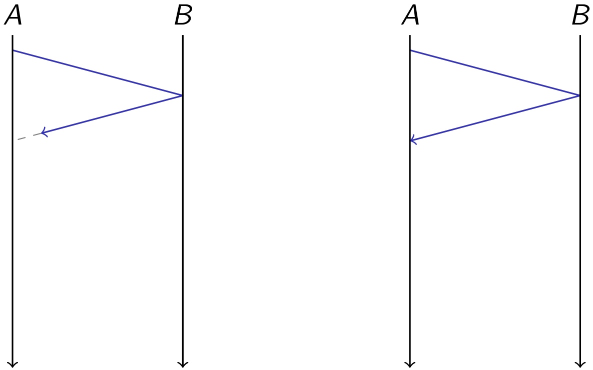
Performance Results

- Tested on 3 machines: USA, Norway, and Denmark
- Results for VIFF 0.4 (VIFF 0.6 is similar or better)
- Tested multiplications and comparisons
- Tested parallel and serial multiplications



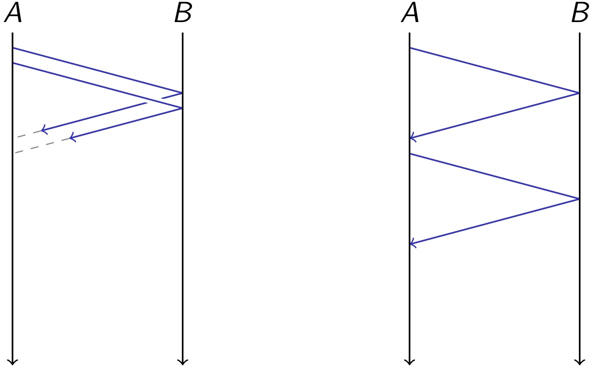
Performance Results

- Tested on 3 machines: USA, Norway, and Denmark
- Results for VIFF 0.4 (VIFF 0.6 is similar or better)
- Tested multiplications and comparisons
- Tested parallel and serial multiplications



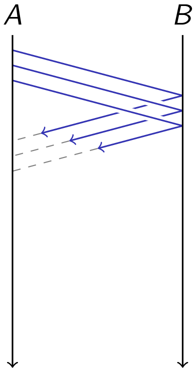
Performance Results

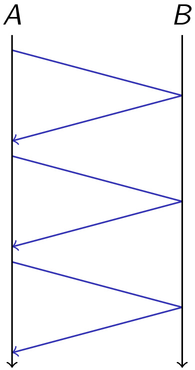
- Tested on 3 machines: USA, Norway, and Denmark
- Results for VIFF 0.4 (VIFF 0.6 is similar or better)
- Tested multiplications and comparisons
- Tested parallel and serial multiplications



Performance Results

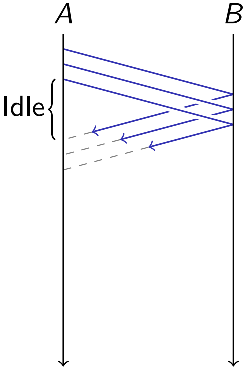
- Tested on 3 machines: USA, Norway, and Denmark
- Results for VIFF 0.4 (VIFF 0.6 is similar or better)
- Tested multiplications and comparisons
- Tested parallel and serial multiplications

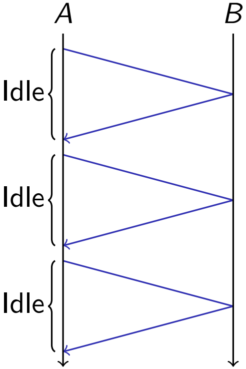




Performance Results

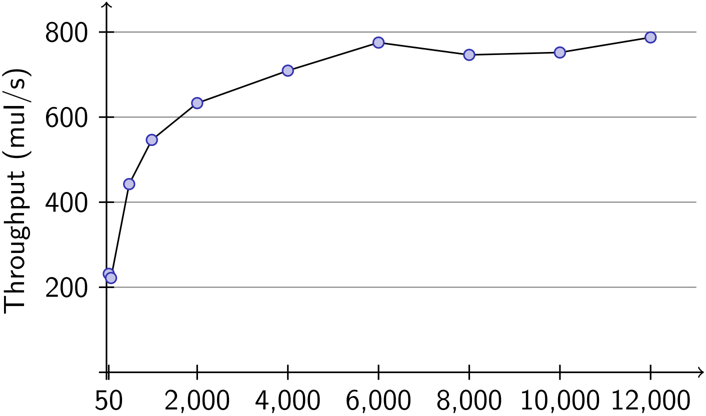
- Tested on 3 machines: USA, Norway, and Denmark
- Results for VIFF 0.4 (VIFF 0.6 is similar or better)
- Tested multiplications and comparisons
- Tested parallel and serial multiplications





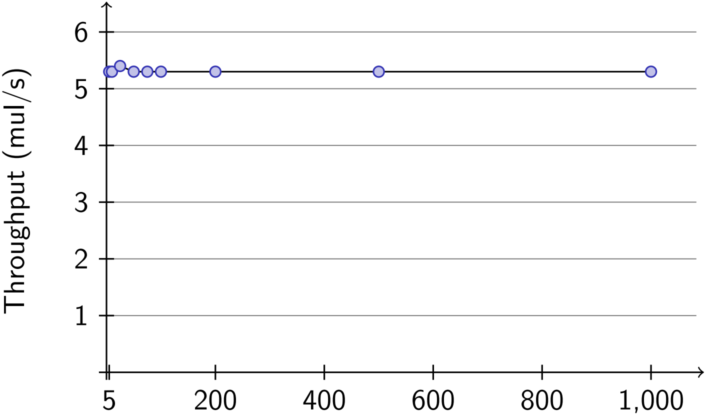
Parallel Multiplications

Multiplying random 65-bit numbers:



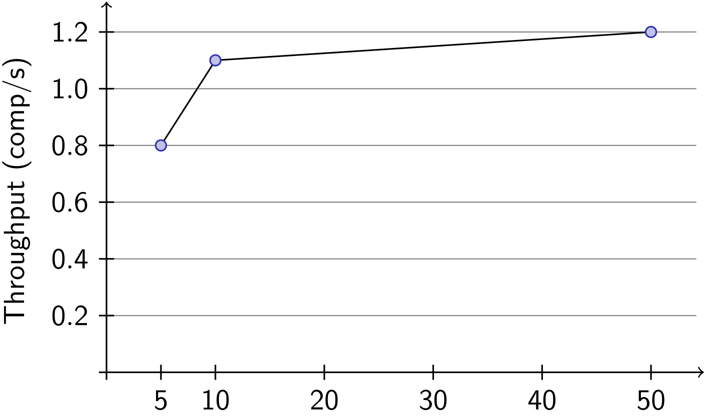
Serial Multiplications

Multiplying random 65-bit numbers:



Parallel Comparisons

Comparing random 32-bit numbers, 65-bit field modulus:



Outline

Background

- Setting
- Related Projects

Design Goals

- Automatic Parallel Execution
- Program Counters

Benchmark Results

- Multiplications
- Comparisons

Possible Improvements

- Program Counters
- Memory Overheads
- Debugging

Conclusion

Program Counter Overheads

- They work, but it's a bit magic. . .
- Exactly when must the program counter be updated?
- Excessive wrapping slows down method calls
- Program counter size depends on stack depth

Memory Overheads

- Python objects have a large memory overhead:
  - Reference count (4 bytes)
  - Object attribute dictionary (144 bytes)
  - Allocation overhead (>8 bytes)
  - ...

Memory Overheads

- Python objects have a large memory overhead:
  - Reference count (4 bytes)
  - Object attribute dictionary (144 bytes)
  - Allocation overhead ($>$8 bytes)
  - ...
- 100,000 field elements with 65-bit prime:
  - Optimal: $\approx$800 KB
  - VIFF: $\approx$40 MB (expanded 50 times)
  - More memory needed for execution tree

Debugging

- Something went wrong! What now?
- Debugging asynchronous programs can be hard

Debugging

- Something went wrong! What now?
- Debugging asynchronous programs can be hard
- Need better
  - Logging infrastructure
  - Handling of exceptions

Type Safety

- Python is a strongly typed language
  - "12" $*$ 3 $==$ "121212" but 12 $*$ 3 $==$ 36
  - "1" $+$ 12 raises TypeError
- Types are only checked at runtime

Type Safety

- Python is a strongly typed language
  - "12" $*$ 3 $==$ "121212" but 12 $*$ 3 $==$ 36
  - "1" $+$ 12 raises TypeError
- Types are only checked at runtime
- Unit tests help here
- Better input validation

Outline

- Background
  - Setting
  - Related Projects
- Design Goals
  - Automatic Parallel Execution
  - Program Counters
- Benchmark Results
  - Multiplications
  - Comparisons
- Possible Improvements
  - Program Counters
  - Memory Overheads
  - Debugging
- Conclusion

Conclusion

- VIFF provides a general framework for MPC
- Automatic parallel execution
- Free Software: LGPL
- Please see `http://viff.dk/`
  - Source code
  - Documentation
  - Mailing list
  - Bibliography

Conclusion

- VIFF provides a general framework for MPC
- Automatic parallel execution
- Free Software: LGPL
- Please see `http://viff.dk/`
  - Source code
  - Documentation
  - Mailing list
  - Bibliography

Thank you! Questions?