Grab-Bag Topics / Demo

Safe Harbor Statement

This presentation may contain forward-looking statements that are based on our current expectations. Forward looking statements may include statements about our financial guidance and expected operating results, our opportunities and future potential, our product development and new product introduction plans, our ability to expand and penetrate our addressable markets and other statements that are not historical facts. These statements are only predictions and actual results may materially vary from those projected. Please refer to Cray's documents filed with the SEC from time to time concerning factors that could affect the Company and these forward-looking statements.

Outline

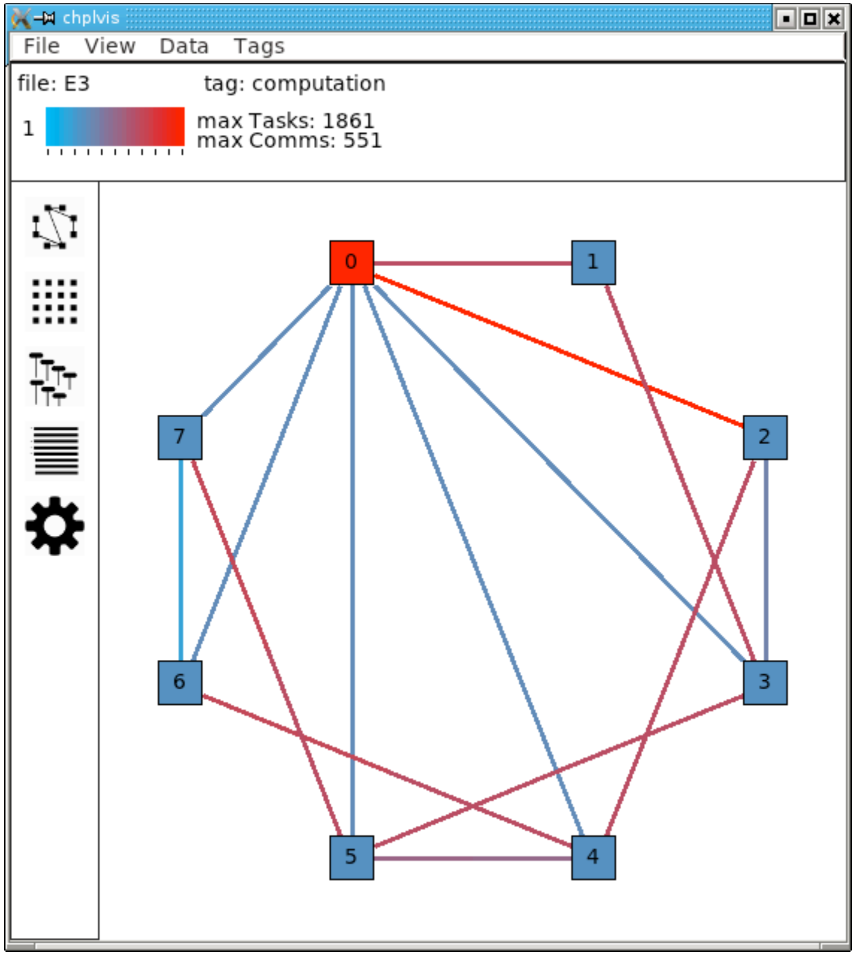
- **Demonstrate chplvis**
- **Study an example: Detecting Duplicate Files**
- **You will learn about:**
  - viewing communication pattern and volume with chplvis
  - optimizing for communication
  - spawning subprocesses with the Spawn module
  - working with the FileSystem and IO modules
  - sorting data with the Sort module
  - calling C functions
- **And use knowledge from earlier:**
  - tuples
  - block distribution
  - zippered iteration
  - forall loops
  - …

chplvis

chplvis

- Seehttp://chapel.cray.com/docs/latest/tools/chplvis/chplvis.html

- Example 3 is Jacobi-like



file: E3

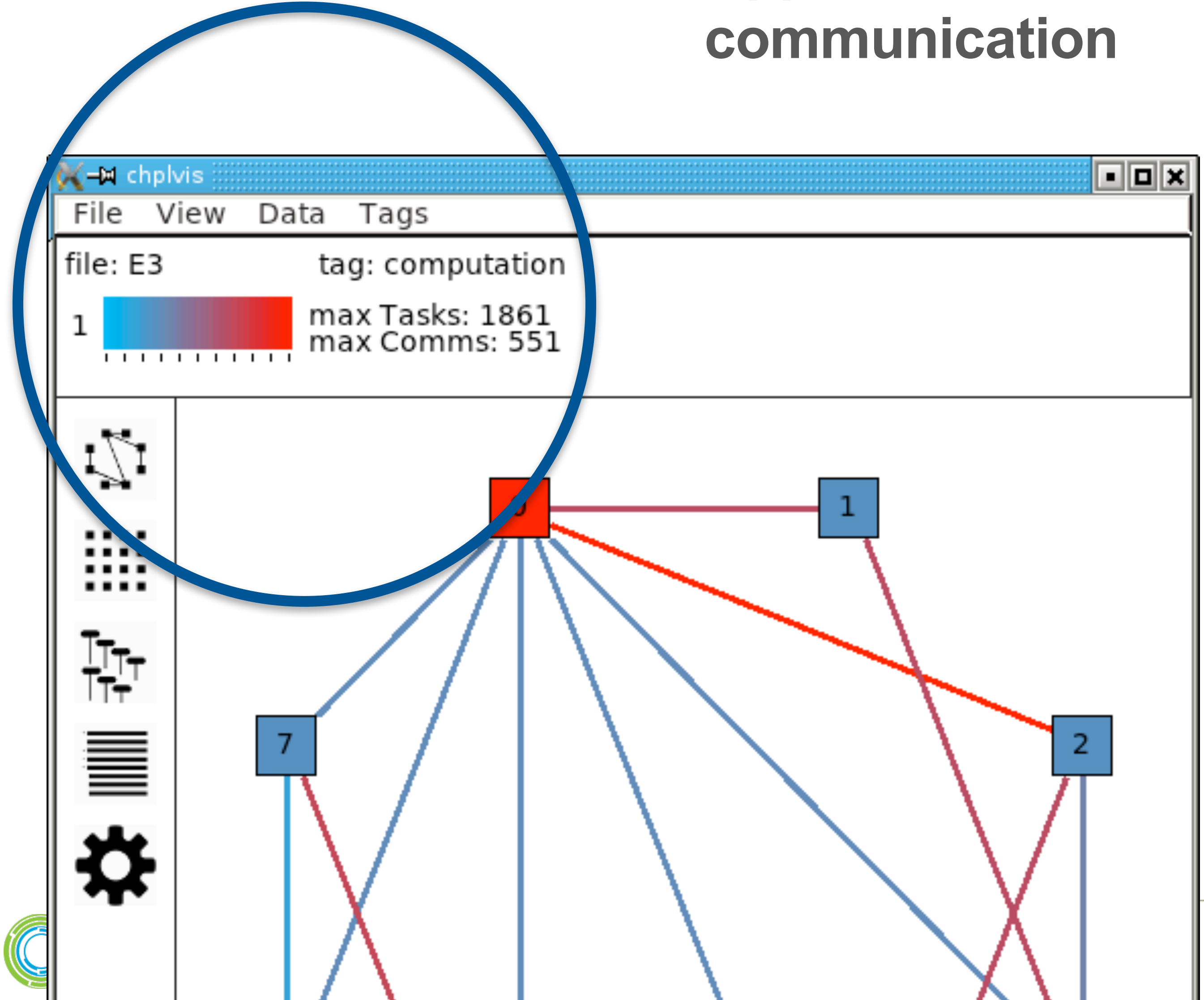
tag: computation

1

max Tasks: 1861
max Comms: 551

chplvis

- Upper Left shows scale of communication



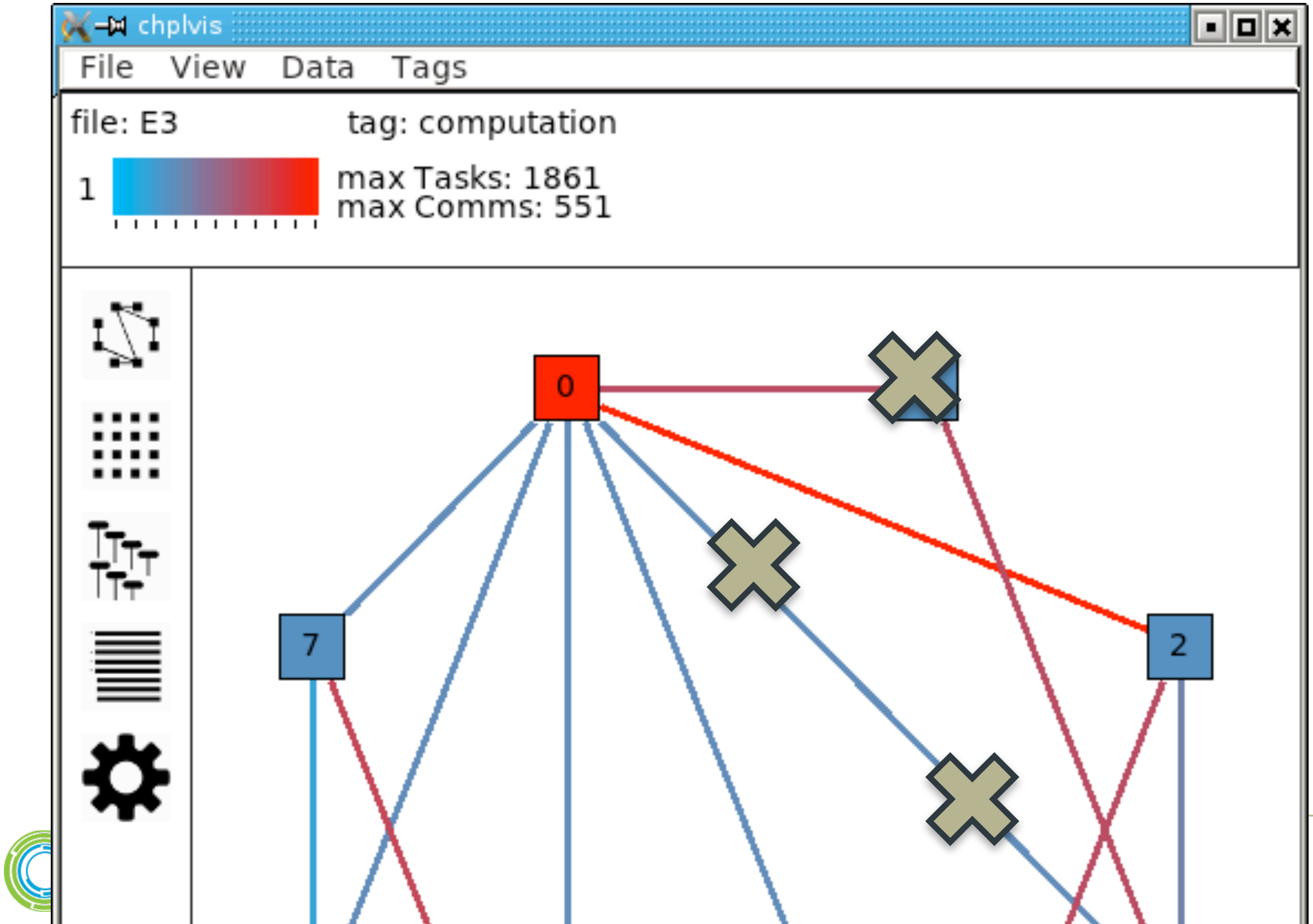
file: E3

tag: computation

max Tasks: 1861
max Comms: 551

chplvis

- **Try clicking on:**
  - both halves of each line
  - the boxes indicating Locales



file: E3 tag: computation

max Tasks: 1861
max Comms: 551

Detecting Duplicate Files

Detecting Duplicate Files

- **Goal: Write a program that produces a list of files that have the same contents**
  - take in files and directories as arguments
  - use SHA1 hash in order to find likely duplicates

Reading Arguments and Enumerating Files

```
proc main(args:[] string)
{
  // This program looks for duplicate files.
  // Arguments are files or directories to include in search.

  // Gather the paths we want to hash to find duplicates.
  // Start out with a 0-length array
  // We'll append to it with push_back
  // This is only possible for arrays that do not share a domain.
  var paths:[1..0] string;

  for arg in args[1..] {
    if isFile(arg) then
      paths.push_back(arg);
    else if isDir(arg) then
      // use FileSystem.findfiles to easily enumerate files.
      // A parallel version is available.
      for path in findfiles(arg, recursive=true) do
        paths.push_back(path);
  }
```

Arrays for the Computation

```
// Create a distributed array of paths so that we can
// distribute the work of hashing files to
// different Locales
var n:int = paths.size;
var BlockN = {1..n} dmapped Block({1..n});
var distributedPaths:[BlockN] string;
distributedPaths = paths;

// Create an array of hashes paths.
// This array is not distributed in this version.
// The array will store (hash, path).
// After computing this array, we'll sort it in order to
// find duplicates.
var hashAndFile:[1..paths.size] (string, string);
```

Computing SHA1 with Spawn

```
// Using the Spawn module, compute the SHA1 sums with an
// external program
forall (id,path) in zip(distributedPaths.domain,
distributedPaths) {
  // The spawn call creates a subprocess. By specifying
  // stdout=PIPE, we are requesting that the output of
  // the subprocess be sent to a pipe that we can read from.
  var sub = spawn(["sha1sum", path], stdout=PIPE);
  // Read the hash value from the output of sha1sum.
  // Note that sha1sum output looks like this:
  // d556d22d3e7b3ae55108442b36b5833523c923b7  file-name
  var hashString:string;
  sub.stdout.read(hashString);
  // Store the hash and the path into the array.
  // Since the array is not distributed, this sends data
  // to Locale 0.
  hashAndFile[id] = (hashString, path);
  sub.wait();
}
```

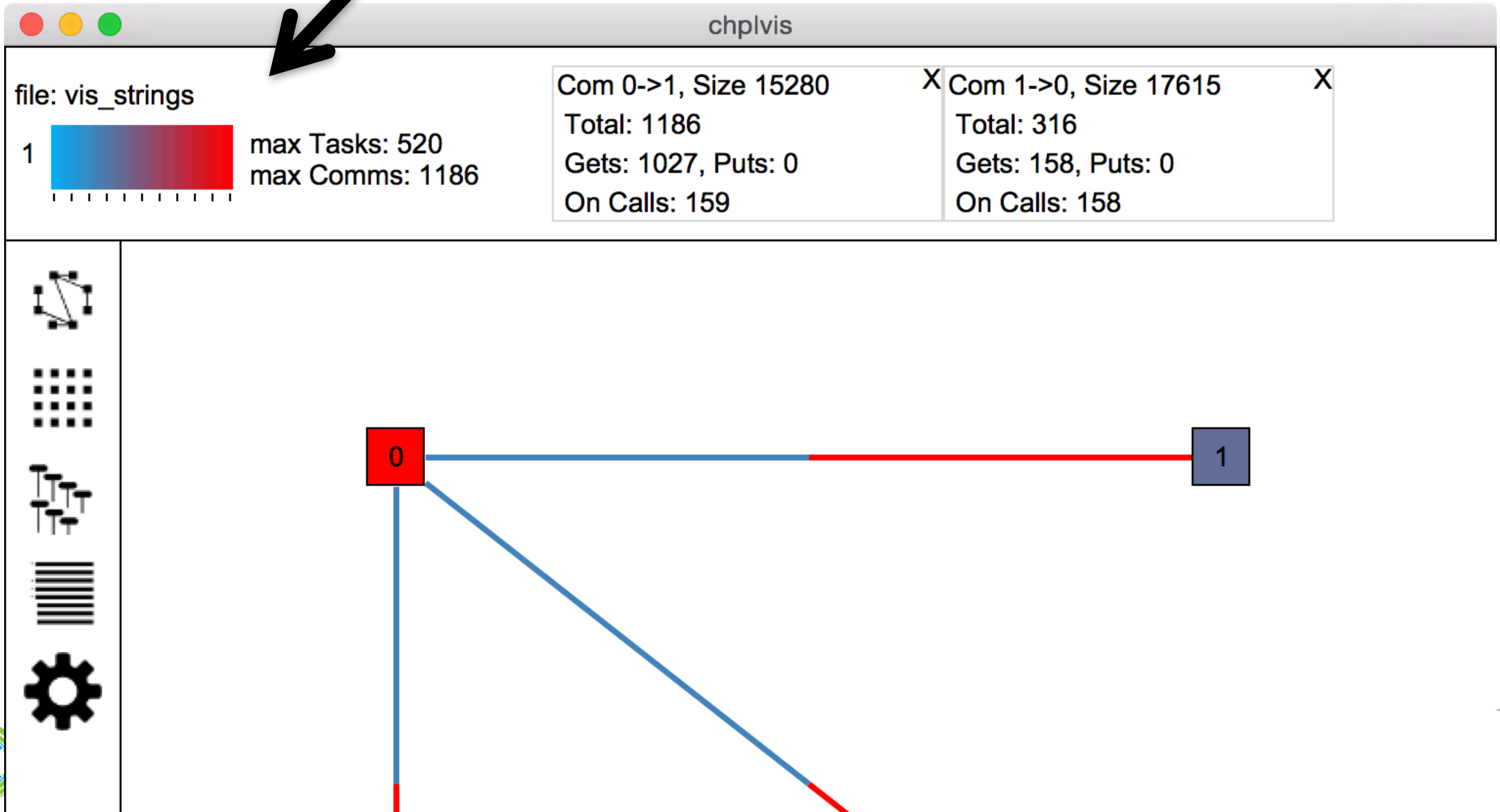
Sorting to Group Duplicates

```
// Sort the hashAndFile array on Locale 0
// Since we stored the hash value first in the tuple elements,
// this call groups values with the same hash.
// Use the Sort Module.
sort(hashAndFile);
```

Let's look at chplvis output!

chplvis output: string version

- Significant communication (for only 316 files)



chplvis

file: vis_strings

1

max Tasks: 520
max Comms: 1186

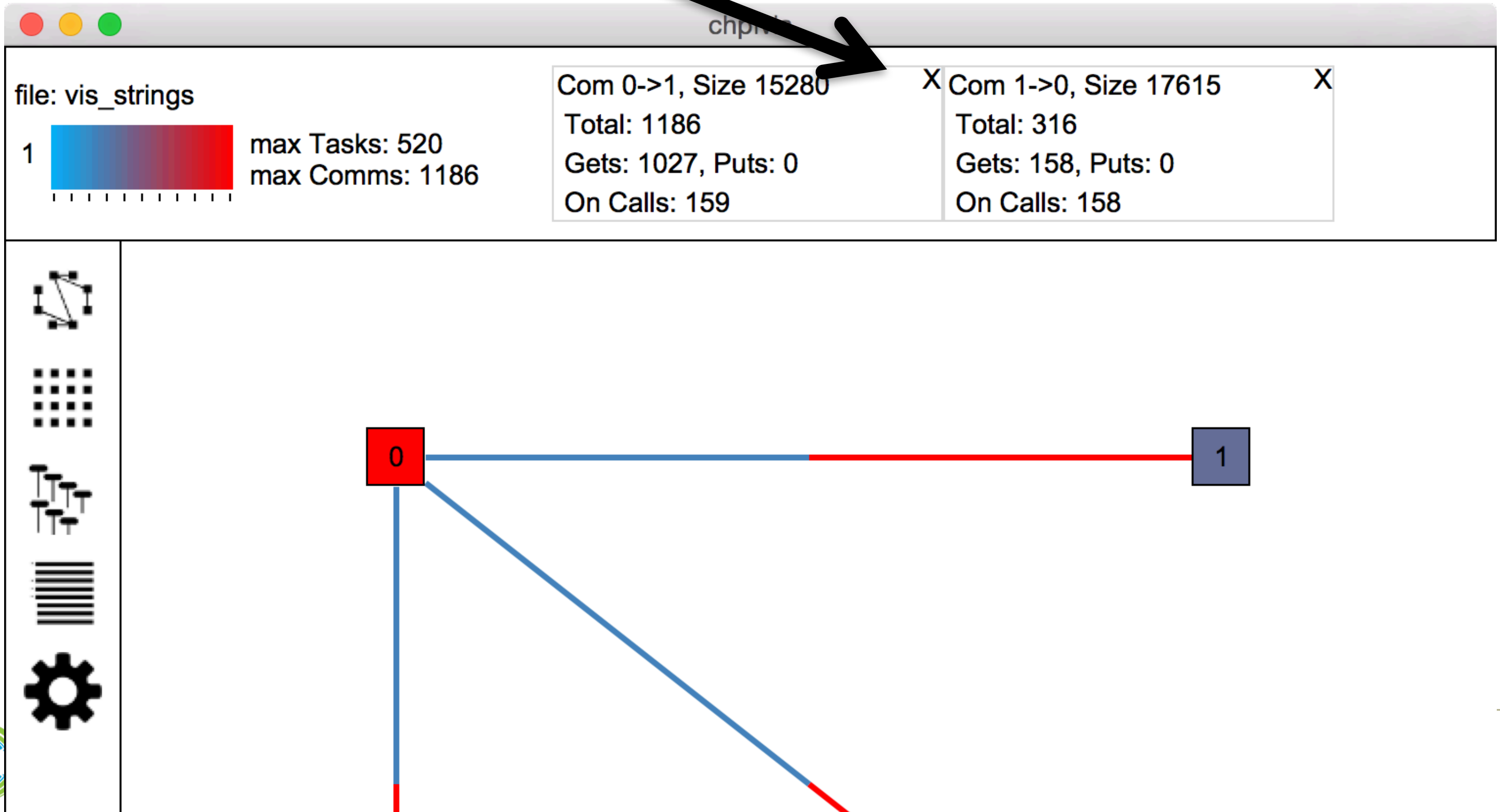
Com 0->1, Size 15280 X
Total: 1186
Gets: 1027, Puts: 0
On Calls: 159

Com 1->0, Size 17615 X
Total: 316
Gets: 158, Puts: 0
On Calls: 158

chplvis output: string version

- Lots of 'on' statements
- communicating strings is expensive!

```
// Store the hash and the path into the array.
// Since the array is not distributed, this sends
// to Locale 0.
hashAndFile[id] = (hashString, path);
```



file: vis_strings

1 max Tasks: 520
max Comms: 1186

Com 0->1, Size 15280 X
Total: 1186
Gets: 1027, Puts: 0
On Calls: 159

Com 1->0, Size 17615 X
Total: 316
Gets: 158, Puts: 0
On Calls: 158

Reducing overhead with integers

Using Integers

- We don't actually need to communicate strings
- Instead of a path string, could store integer index into paths array
- Instead of a hash string, could store a tuple of integers
  - SHA1 hash is 20 bytes -- fits in 3 Chapel ints

Creating a type for hashes

```
// a SHA-1 hash is 160 bits, so it fits in 3 64-bit ints.
type Hash = (int,int,int);
```

Using integers in the hashAndFile array

```
// Create an array of hashes and file ids
// a file id is just the index into the paths array.
var hashAndFileId:[1..paths.size] (Hash, int);
```

Working with integers in the loop

```
var hash = stringToHash(hashString);
// This version is just communicating 4 integer values
// back to Locale 0.
hashAndFileId[id] = (hash, id);
```

Converting hex to ints

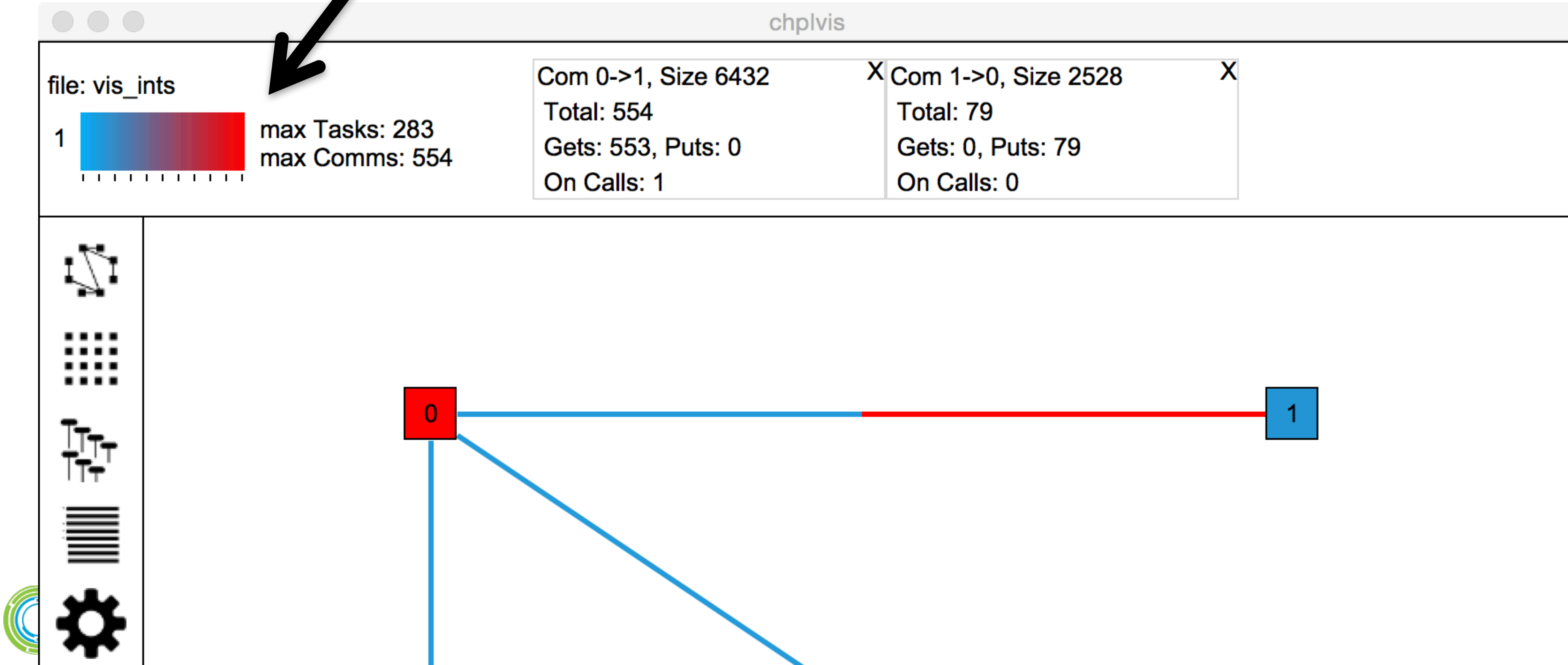
```
proc stringToHash(s:string): Hash {
  // The below is a workaround since Chapel doesn't yet have
  // an equivalent of sscanf in C and readf for integers
  // can't take in a maximum field width

  // Open up an in-memory "file"
  var f = openmem();
  var w = f.writer();
  // Write int-sized substrings separated by spaces
  w.write(s[1..16], " ");
  w.write(s[17..32], " ");
  w.write(s[17..32]);
  w.close();
  var r = f.reader();
  var hash:Hash;
  // Use Formatted I/O to read hex values into integers
  r.readf("%xu%xu%xu", hash(1), hash(2), hash(3));
  r.close();
  return hash;
}
```

Let's look at chplvis output!

chplvis output: integer version

- Reduced communication (for only 316 files)



chplvis

file: vis_ints

1

max Tasks: 283
max Comms: 554

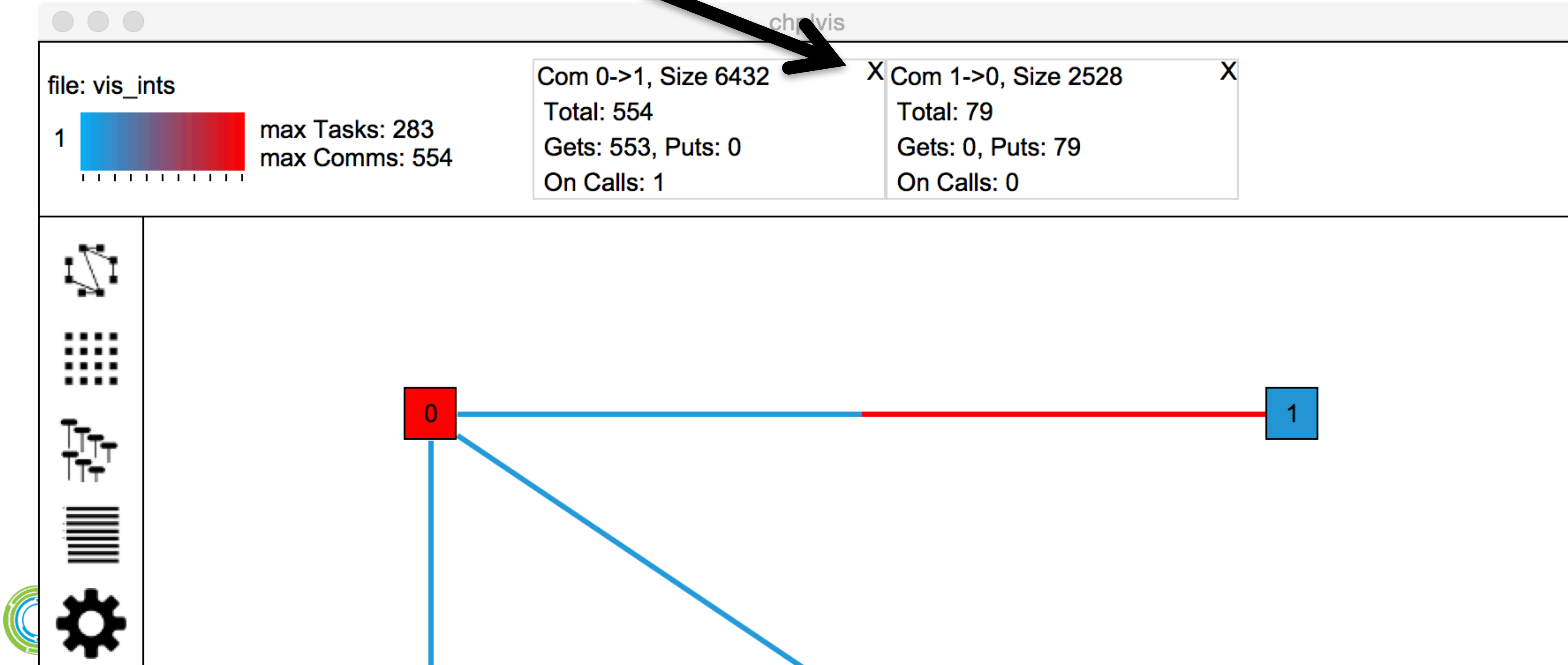
Com 0->1, Size 6432 X
Total: 554
Gets: 553, Puts: 0
On Calls: 1

Com 1->0, Size 2528 X
Total: 79
Gets: 0, Puts: 79
On Calls: 0

chplvis output: integer version

- Only 1 on statement
- Now communication to Locale 0 uses PUT

```
// This version is just communicating 4 integers
// back to Locale 0.
hashAndFileId[id] = (hash, id);
```



chplvis

file: vis_ints

1

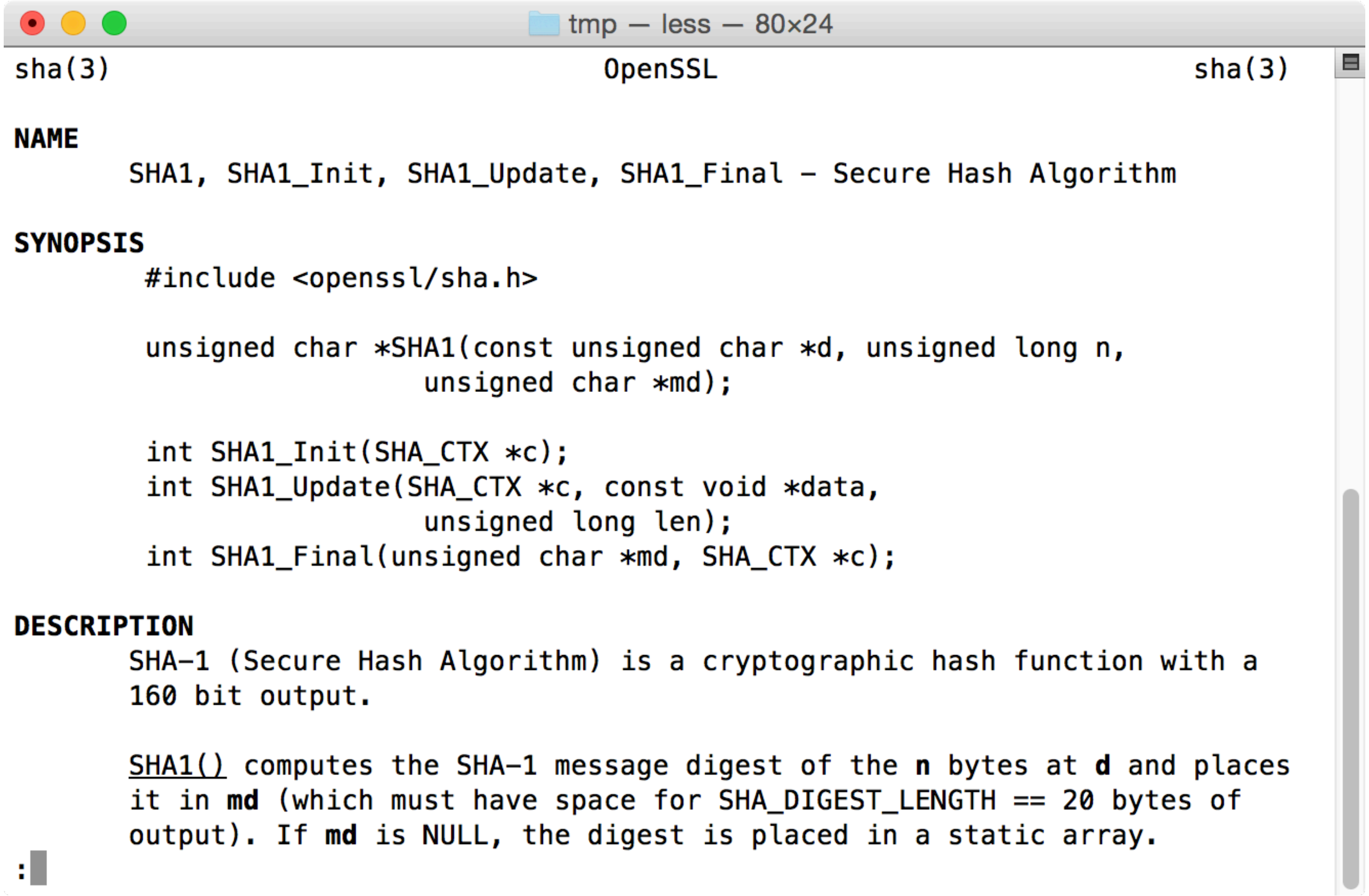
max Tasks: 283
max Comms: 554

Com 0->1, Size 6432 X
Total: 554
Gets: 553, Puts: 0
On Calls: 1

Com 1->0, Size 2528 X
Total: 79
Gets: 0, Puts: 79
On Calls: 0

Using a C library to SHA

SHA1 available in OpenSSL library



```
sha(3)                              OpenSSL                              sha(3)

NAME
       SHA1, SHA1_Init, SHA1_Update, SHA1_Final - Secure Hash Algorithm

SYNOPSIS
        #include <openssl/sha.h>

        unsigned char *SHA1(const unsigned char *d, unsigned long n,
                         unsigned char *md);

        int SHA1_Init(SHA_CTX *c);
        int SHA1_Update(SHA_CTX *c, const void *data,
                         unsigned long len);
        int SHA1_Final(unsigned char *md, SHA_CTX *c);

DESCRIPTION
       SHA-1 (Secure Hash Algorithm) is a cryptographic hash function with a
       160 bit output.

       SHA1() computes the SHA-1 message digest of the n bytes at d and places
       it in md (which must have space for SHA_DIGEST_LENGTH == 20 bytes of
       output). If md is NULL, the digest is placed in a static array.
:
```

Including SHA1

```
// This require statement allows this module to add
// some required libraries to the link line
require "-lcrypto", "-lssl";

// The extern block allows Chapel source code to include
// C declarations. The declarations are automatically
// added to the enclosing Chapel scope. Functions,
// variables, and types are supported - including
// inline functions. Macros have limited support.
// See C Interoperability
extern {
  #include <openssl/sha.h>
}
```

Calling SHA1

```
// The extern block above included everything in
// openssl/sha.h, including the SHA1 function. But,
// in order to call it, we need to create C types
// from some Chapel data.
//   string.c_str() returns a C string referring to
//                  the string's data
//   c_ptrTo(something) returns a C pointer referring
//                      to something
SHA1(data.c_str(), data.length:uint, c_ptrTo(mdArray));
```

Alternative way of including SHA1

```
// This require statement indicates that the generated code
// should #include "openssl/sha.h" and be compiled with
// -lcrypto -lssl
require "openssl/sha.h", "-lcrypto", "-lssl";
// This 'extern proc' declaration tells the Chapel
// compiler that a C function SHA1 is available and
// describes the arguments in the Chapel type system.
extern proc SHA1(d:c_string, n:size_t, md:c_ptr(uint(8)));
```

Legal Disclaimer

Information in this document is provided in connection with Cray Inc. products. No license, express or implied, to any intellectual property rights is granted by this document.

Cray Inc. may make changes to specifications and product descriptions at any time, without notice.

All products, dates and figures specified are preliminary based on current expectations, and are subject to change without notice.

Cray hardware and software products may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Cray uses codenames internally to identify products that are in development and not yet publically announced for release. Customers and other third parties are not authorized by Cray Inc. to use codenames in advertising, promotion or marketing and any use of Cray Inc. internal codenames is at the sole risk of the user.

Performance tests and ratings are measured using specific systems and/or components and reflect the approximate performance of Cray Inc. products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance.

The following are trademarks of Cray Inc. and are registered in the United States and other countries: CRAY and design, SONEXION, and URIKA. The following are trademarks of Cray Inc.: ACE, APPRENTICE2, CHAPEL, CLUSTER CONNECT, CRAYPAT, CRAYPORT, ECOPHLEX, LIBSCI, NODEKARE, THREADSTORM. The following system family marks, and associated model number marks, are trademarks of Cray Inc.: CS, CX, XC, XE, XK, XMT, and XT. The registered trademark LINUX is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis. Other trademarks used in this document are the property of their respective owners.