CPSC 4240/5240: Parallel Programming Techniques

Lecture 12: MPI, Data Representations, Graphs

Lecturer: Quanquan C. Liu

Spring 2026

Announcements

- Homework 2: we'll increase the number of threads we'll use for the performance test to **8**

Announcements

- Homework 2: we'll increase the number of threads we'll use for the performance test to **8**
- **Midterm 2 moved to March 24th**
  - More difficult than first midterm
  - Will focus on newer material (covered after the first midterm)

Announcements

- Homework 2: we'll increase the number of threads we'll use for the performance test to **8**
- **Midterm 2 moved to March 24th**
  - More difficult than first midterm
  - Will focus on newer material (covered after the first midterm)
- Homework 3 will be released on Thursday this week

Final Project Coming Up After Break!

- See Syllabus for a description
- 1-3 people
- You will be spending April working on your project
- 20% of your grade
- Final project presentation last two days of class

Send and Receive Example Program

```
#include <iostream>
#include <cassert>

int main(int argc, char* argv[]) {
   int N = 32;
   double fibs[N + 2];

   // Initialize the first two Fibonacci values
   fibs[0] = 1;
   fibs[1] = 1;

   // Compute and print the sequence
   for (int i = 2; i < N; i++) {
      fibs[i] = fibs[i - 1] + fibs[i - 2];
      std::cout << "The " << i << "th Fibonacci number is "
                << fibs[i] << "." << std::endl;
   }

   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Status status;
    // Starting message: first two Fibonacci numbers.
    double msg[2] = {1, 1};

    // For processes with rank > 0, receive a message from any
source.
    if (rank > 0) {
        double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Compute the next Fibonacci number.
        double next = fibs[0] + fibs[1];
        // Update the message: the new pair is (previous second,
next).
        msg[0] = fibs[1];
        msg[1] = next;

        // Output the Fibonacci number corresponding to this
process.
        std::cout << "The " << (rank + 2)
                  << "th Fibonacci number is " << next << ".\n";
    }

    // If there is a next process, send the updated message.
    if (rank + 1 < size) {
        int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
        assert(ret == MPI_SUCCESS);
    }

    // Finalize the MPI environment.
    MPI_Finalize();
    return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

    int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD,
&size);
   MPI_Comm_rank(MPI_COMM_WORLD,
&rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
            << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any source.
   if (rank > 0) {
      double fibs[2];
```

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

```
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second, next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
         msg[0] = fibs[1];
         msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
            << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second, next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE,
rank + 1, 0, MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_C
    MPI_Comm_rank(MPI_

    MPI_Status status;
    // Starting message: firs
    double msg[2] = {1, 1};

    // For processes with ra
source.
    if (rank > 0) {
        double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Compute the next Fibonacci number.
        double next = fibs[0] + fibs[1];
        // Update the message: the new pair is (previous second,
next).
        msg[0] = fibs[1];
        msg[1] = next;
```

```
orresponding to this

' << next << ".\n";

e updated message.

OUBLE, rank + 1, 0,
```

```
    // Finalize the MPI environment.
    MPI_Finalize();
    return 0;
}
```

```
mpirun -np 8 ./fibonacci

The 3th Fibonacci number is 2.
The 4th Fibonacci number is 3.
The 5th Fibonacci number is 5.
The 6th Fibonacci number is 8.
The 7th Fibonacci number is 13.
The 8th Fibonacci number is 21.
The 9th Fibonacci number is 34.
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    MPI_Status status;
    // Starting message: first two Fibonacci numbers.
    double msg[2] = {1, 1};

    //
sour
    if

MPI
MPI
```

Could result in deadlock in certain situations!

```
        // Compute the next Fibonacci number.
        double next = fibs[0] + fibs[1];
        // Update the message: the new pair is (previous second,
next).
        msg[0] = fibs[1];
        msg[1] = next;

        // Output the Fibonacci number corresponding to this
process.
        std::cout << "The " << (rank + 2)
                  << "th Fibonacci number is " << next << ".\n";
    }

    // If there is a next process, send the updated message.
    if (rank + 1 < size) {
        int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
_COMM_WORLD);
        assert(ret == MPI_SUCCESS);


    Finalize the MPI environment.
    MPI_Finalize();
    turn 0;
}
```

Example of blocking in MPI

When can operations block each other?

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);

   MPI_Status status;
   // Starting message: first two Fibonacci numbers.
   double msg[2] = {1, 1};

   // For processes with rank > 0, receive a message from any
source.
   if (rank > 0) {
      double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
process.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";
   }

   // If there is a next process, send the updated message.
   if (rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
MPI_COMM_WORLD);
      assert(ret == MPI_SUCCESS);
   }

   // Finalize the MPI environment.
   MPI_Finalize();
   return 0;
}
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
    MPI_Init(&argc, &argv);

    int rank, size;
    MPI_Comm_size(MPI_C
    MPI_Comm_rank(MPI_

    MPI_Status status;
    // Starting message: firs
    double msg[2] = {1, 1};

    // For processes with ra
source.
    if (rank > 0) {
        double fibs[2];

MPI_Recv(fibs, 2, MPI_DOUBLE, rank - 1, 0,
MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        // Compute the next Fibonacci number.
        double next = fibs[0] + fibs[1];
        // Update the message: the new pair is (previous second,
next).
        msg[0] = fibs[1];
        msg[1] = next;
```

```
orresponding to this

' << next << ".\n";

e updated message.

OUBLE, rank + 1, 0,
```

```
    // Finalize the MPI environment.
    MPI_Finalize();
    return 0;
}
```

```
mpirun -np 8 ./fibonacci

The 3th Fibonacci number is 2.
The 4th Fibonacci number is 3.
The 5th Fibonacci number is 5.
The 6th Fibonacci number is 8.
The 7th Fibonacci number is 13.
The 8th Fibonacci number is 21.
The 9th Fibonacci number is 34.
```

```
#include <mpi.h>
#include <cassert>
#include <iostream>

int main(int argc, char* argv[]) {
   MPI_Init(&argc, &argv);

   int rank, size;
   MI
   MI

   MI
   //
   do

   //
sour
   if (


MPI_
MPI_
```

```
      // Compute the next Fibonacci number.
      double next = fibs[0] + fibs[1];
      // Update the message: the new pair is (previous second,
next).
      msg[0] = fibs[1];
      msg[1] = next;

      // Output the Fibonacci number corresponding to this
ess.
      std::cout << "The " << (rank + 2)
                << "th Fibonacci number is " << next << ".\n";


      If there is a next process, send the updated message.
      rank + 1 < size) {
      int ret = MPI_Send(msg, 2, MPI_DOUBLE, rank + 1, 0,
      _COMM_WORLD);
      assert(ret == MPI_SUCCESS);


      Finalize the MPI environment.
      MPI_Finalize();
      turn 0;
}
```

Could result in deadlock in certain situations!

Briefly discuss next class

Sources of Deadlocks

- First, send a large message from process 0 to process 1

Sources of Deadlocks

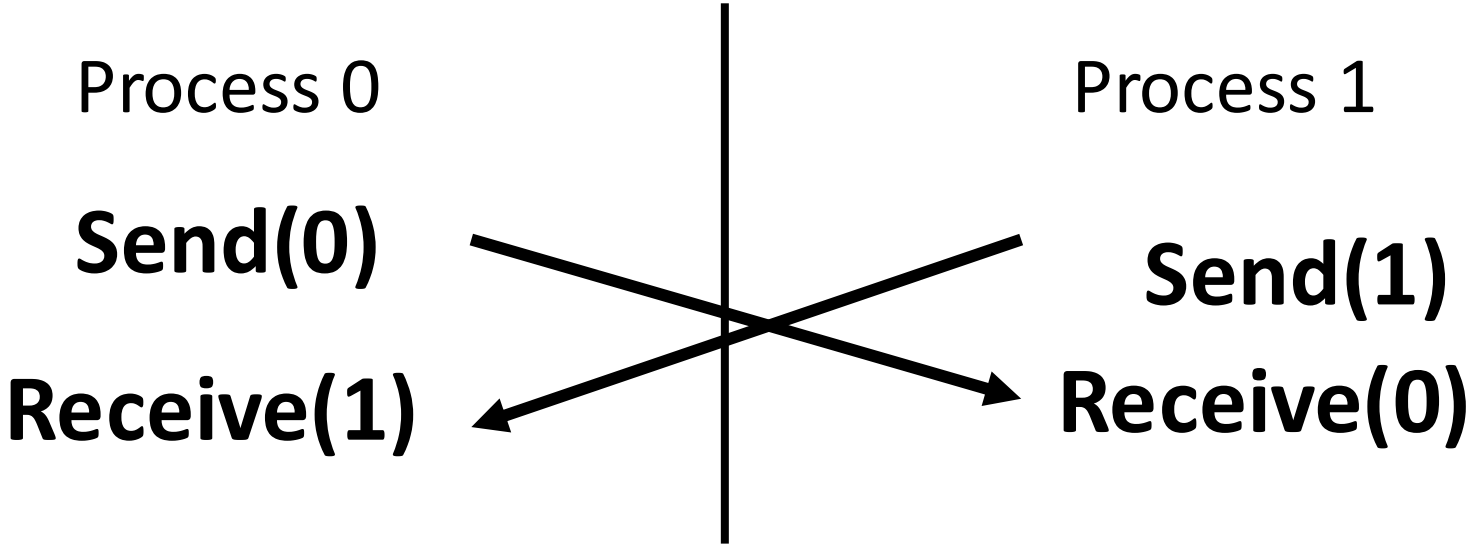
- First, send a large message from process 0 to process 1
  - Suppose there is insufficient storage at the destination (process 1)

Sources of Deadlocks

- First, send a large message from process 0 to process 1
  - Suppose there is insufficient storage at the destination (process 1)
  - Then, send must wait for the user to provide memory space

Sources of Deadlocks

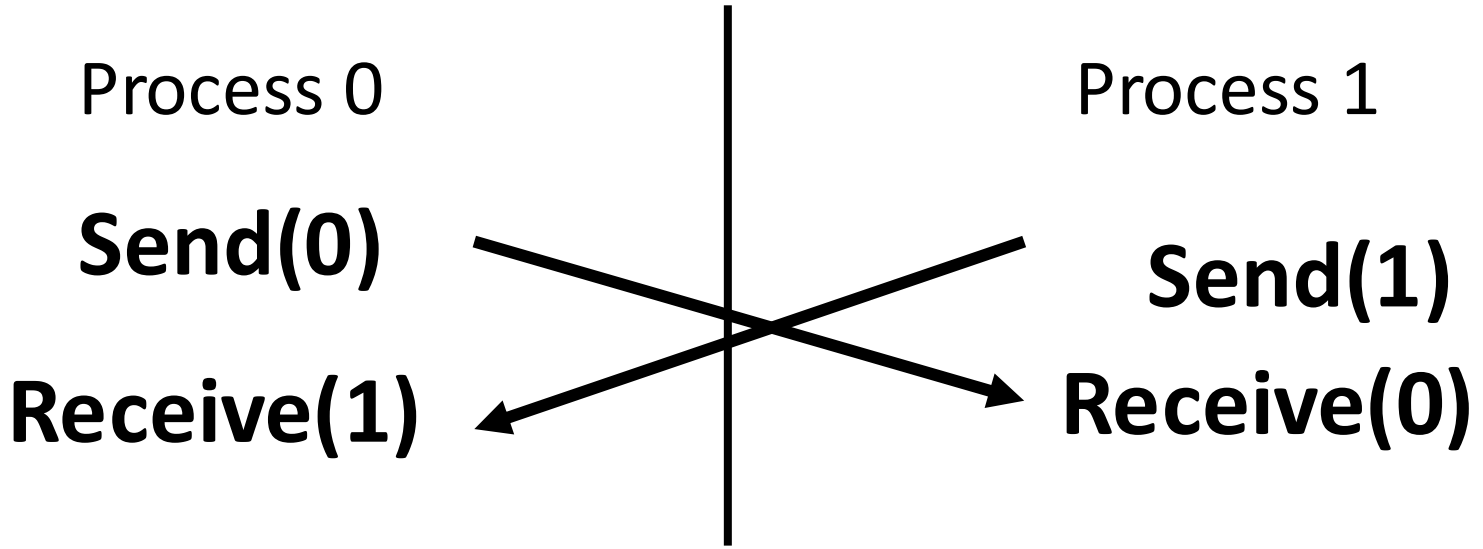
- First, send a large message from process 0 to process 1
  - Suppose there is insufficient storage at the destination (process 1)
  - Then, send must wait for the user to provide memory space



Sources of Deadlocks

For **small** messages, send messages can be posted to the buffer before receiver receives the message

- First, send a large message from p
  - Suppose there is insufficient sto (process 1)
  - Then, send must wait for the user to provide memory space



For **large** messages, receiver needs to start receiving before send can unblock. **Here, both 0 and 1 could send large messages!**

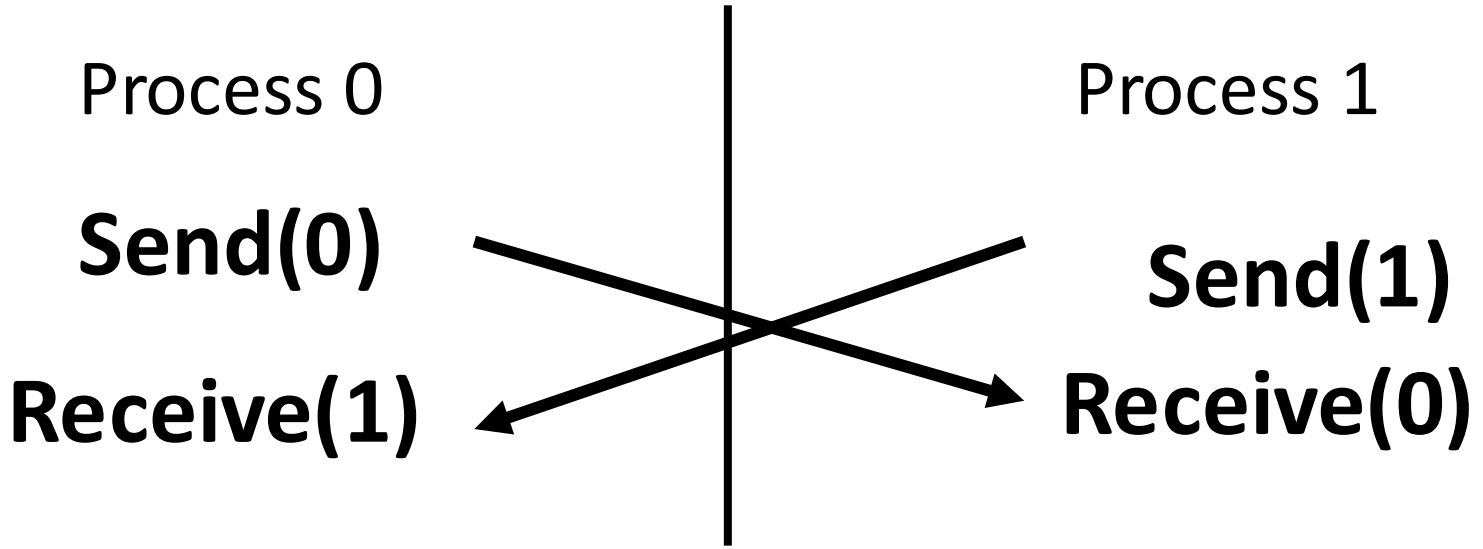
For **small** messages, send messages can be posted to the buffer before receiver receives the message

cks

rom p

nt sto

- Then, send must wait for the user to provide memory space

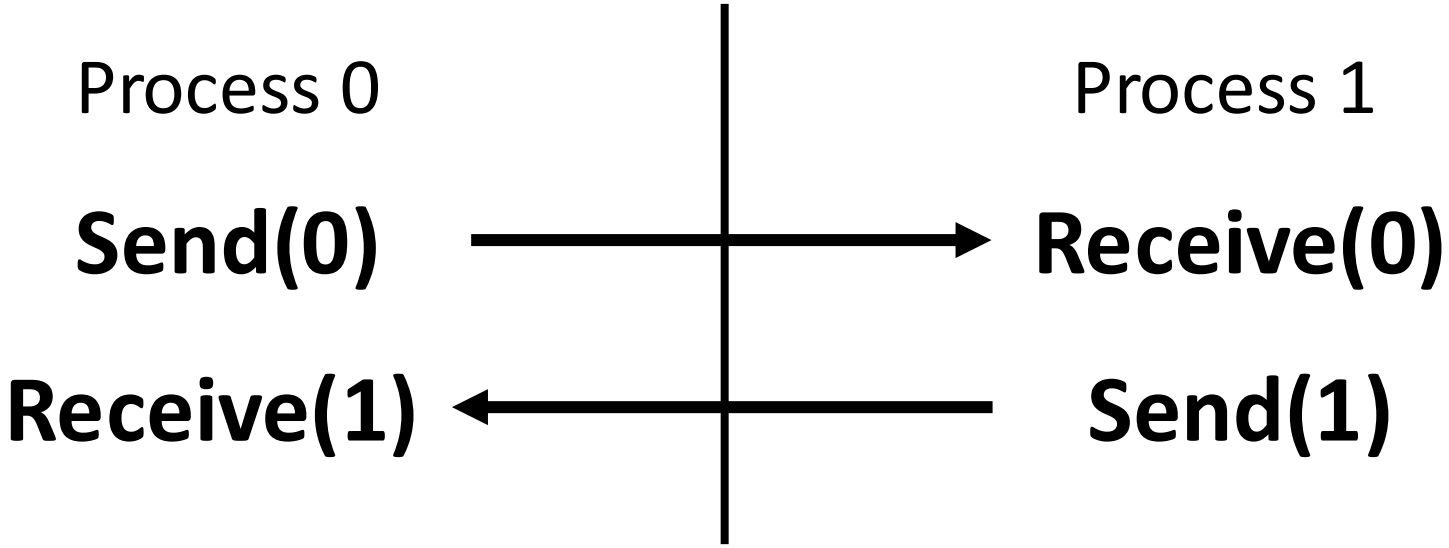


Solutions to Deadlocks

- Order the operations more carefully:

Solutions to Deadlocks

- Order the operations more carefully:



Solutions to Deadlocks

- Use non-blocking send and receive operations:

Solutions to Deadlocks

- Use non-blocking send and receive operations:

Process 0	Process 1
Isend(1)	**Isend(0)**
Irecv(0)	**Irecv(1)**
Waitall	**Waitall**

Non-Blocking Receive and Send

```
int MPI_Isend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

const void *buf:

Pointer to data you want to send

Non-Blocking Receive and Send

```
int MPI_Isend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

const void *buf:

Pointer to data you want to send

int count:

Number of elements in data you want to send

Non-Blocking Receive and Send

```
int MPI_Isend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

const void *buf:

Pointer to data you want to send

int count:

Number of elements in data you want to send

datatype:

Type of data in the buffer

Non-Blocking Receive and Send

```
int MPI_Isend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

const void *buf:

Pointer to data you want to send

int count:

Number of elements in data you want to send

datatype:

Type of data in the buffer

int dest:

The rank of the destination process

Non-Blocking Receive and Send

```
int MPI_Isend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

int tag:

User-defined integer that labels the message

Non-Blocking Receive and Send

```
int MPI_Isend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

int tag:

User-defined integer that labels the message

MPI_Comm comm:

Communicator

Non-Blocking Receive and Send

```
int MPI_Isend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

int tag:
User-defined integer that labels the message

MPI_Comm comm:
Communicator

MPI_Request *request:
Pointer to an MPI_Request handle which is used to track the status of non-blocking operations.

Non-Blocking Receive and Send

```
int MPI_Isend(const void *buf,
              int count,
              MPI_Datatype datatype,
              int dest,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

```
int MPI_Irecv(void *buf,
              int count,
              MPI_Datatype datatype,
              int source,
              int tag,
              MPI_Comm comm,
              MPI_Request *request);
```

Non-Blocking Receive and Send

```
MPI_Request request;
MPI_Isend(buffer, count, datatype, dest, tag, comm, &request);

MPI_Wait(&request, MPI_STATUS_IGNORE);
```

Non-Blocking Receive and Send

```
MPI_Request request;
MPI_Isend(buffer, count, datatype, dest, tag, comm, &request);

MPI_Wait(&request, MPI_STATUS_IGNORE);
```

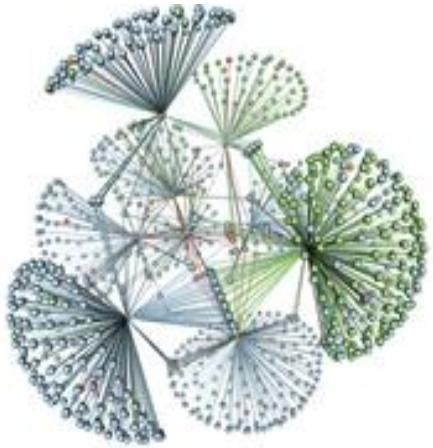
MPI_Wait waits for the operation to complete before moving on in the code

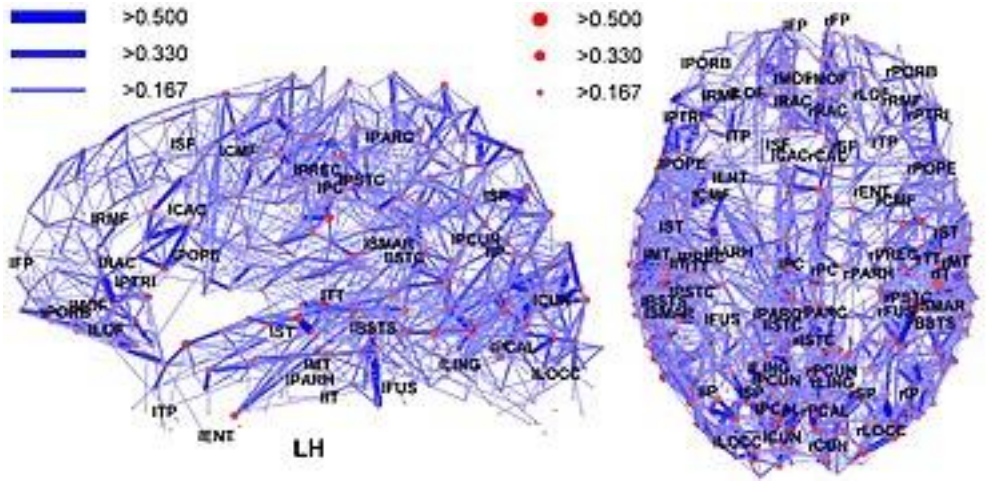
Parallel Graph Algorithms

Graphs are everywhere in today's real world

Why Graphs?

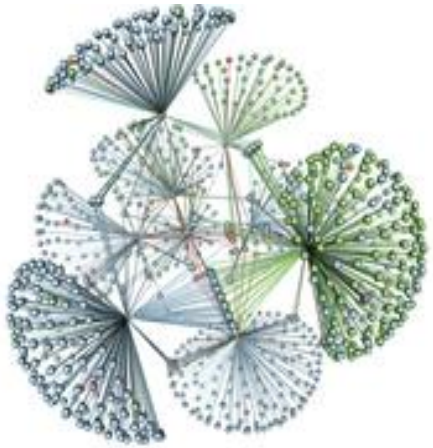
- Graphs model **relationships between objects**
  - **Nodes** are **objects**
  - **Edges** are **relationships**

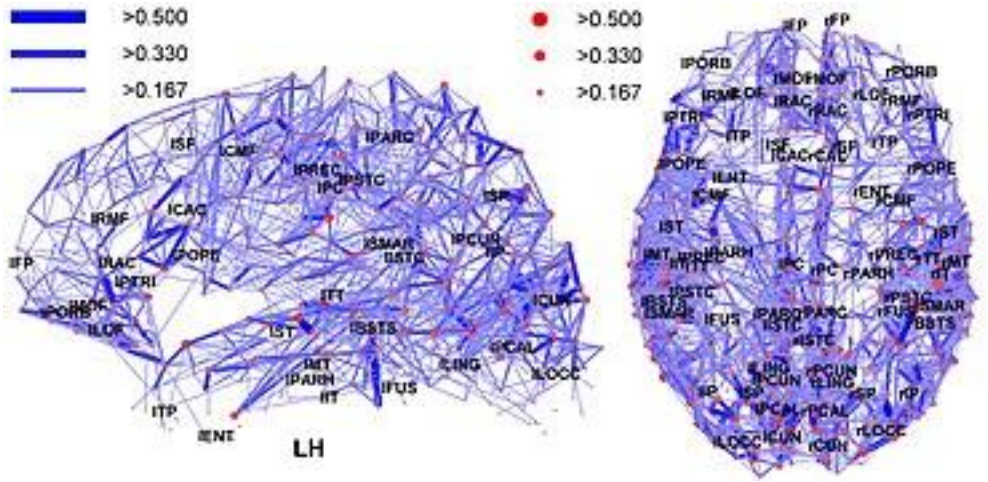




Why Graphs?

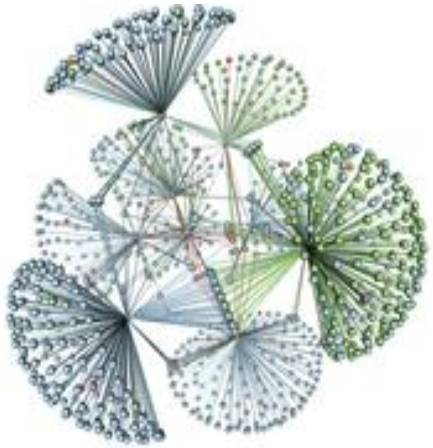
- Graphs model **relationships between objects**
  - **Nodes** are **objects**
  - **Edges** are **relationships**
- Many **applications**:
  - Social networks
  - Transportation
  - Financial networks
  - Protein interaction networks
  - Neuroscience
  - Database operations

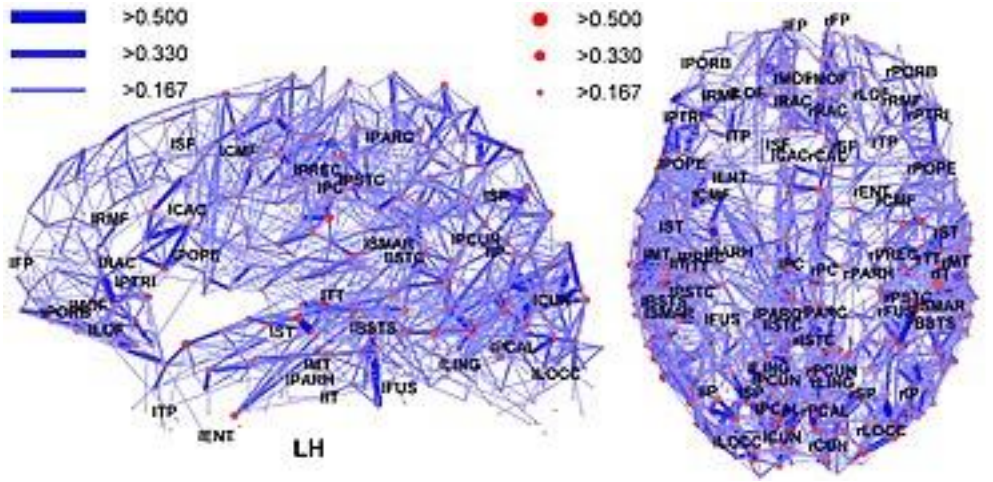




Why Graph Algorithms?

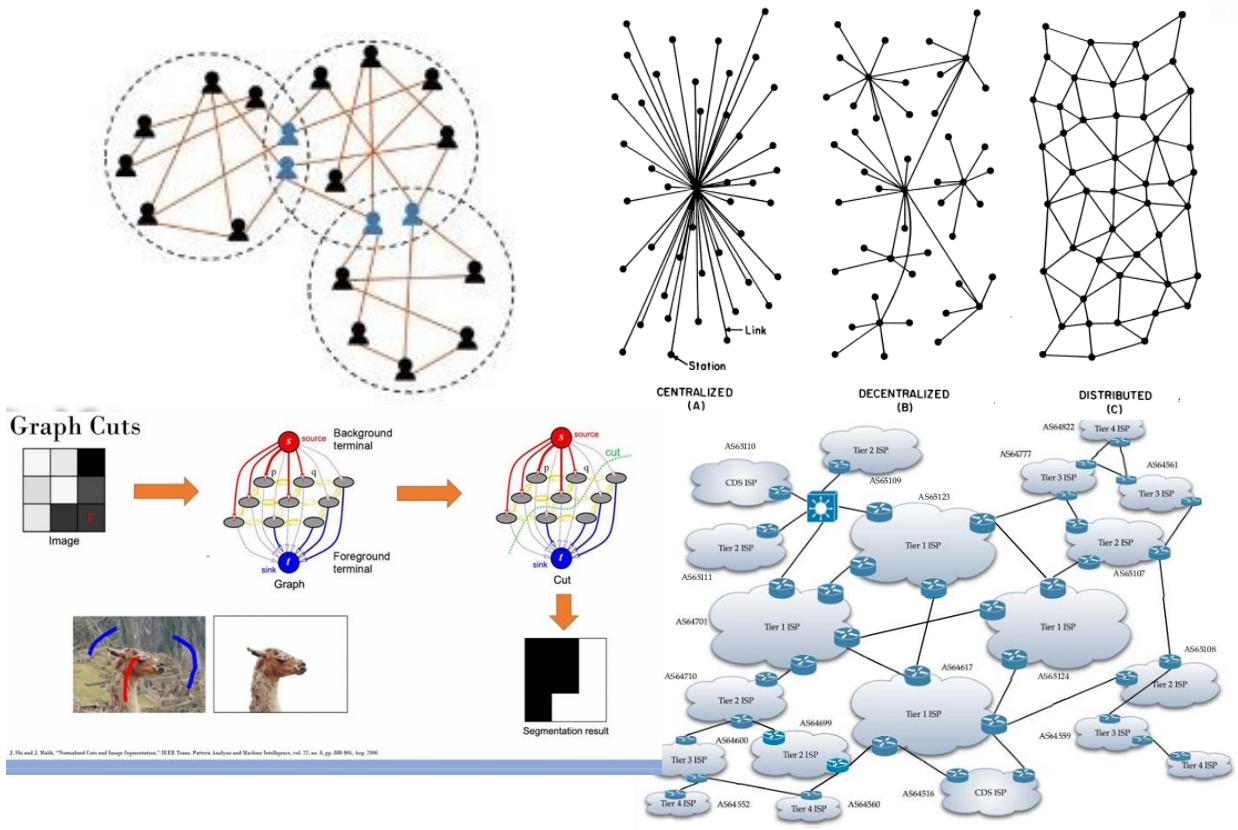
- Graphs model **relationships between objects**
  - **Nodes** are **objects**
  - **Edges** are **relationships**
- Many **applications**:
  - Social networks
  - Transportation
  - Financial networks
  - Protein interaction networks
  - Neuroscience
  - Database operations
- **Useful** and **interesting analytics**





Useful and Interesting Analytics

- Community detection in social networks
- Image processing
- Finding weak spots in data and communication networks
- IP traffic routing
- Spam and fraud detection
- Ecological network analysis
- Recommendation systems



Many Modern Challenges: **Massive** Datasets

MASSIVE graph data sets

~ 27 billion comments

ClueWeb

~ 10 billion edges

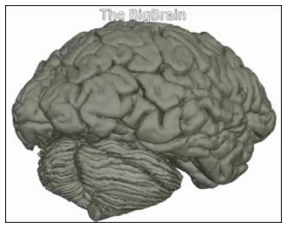
Google

~ 6 trillion edges

Web Data Commons Hyperlink Graph

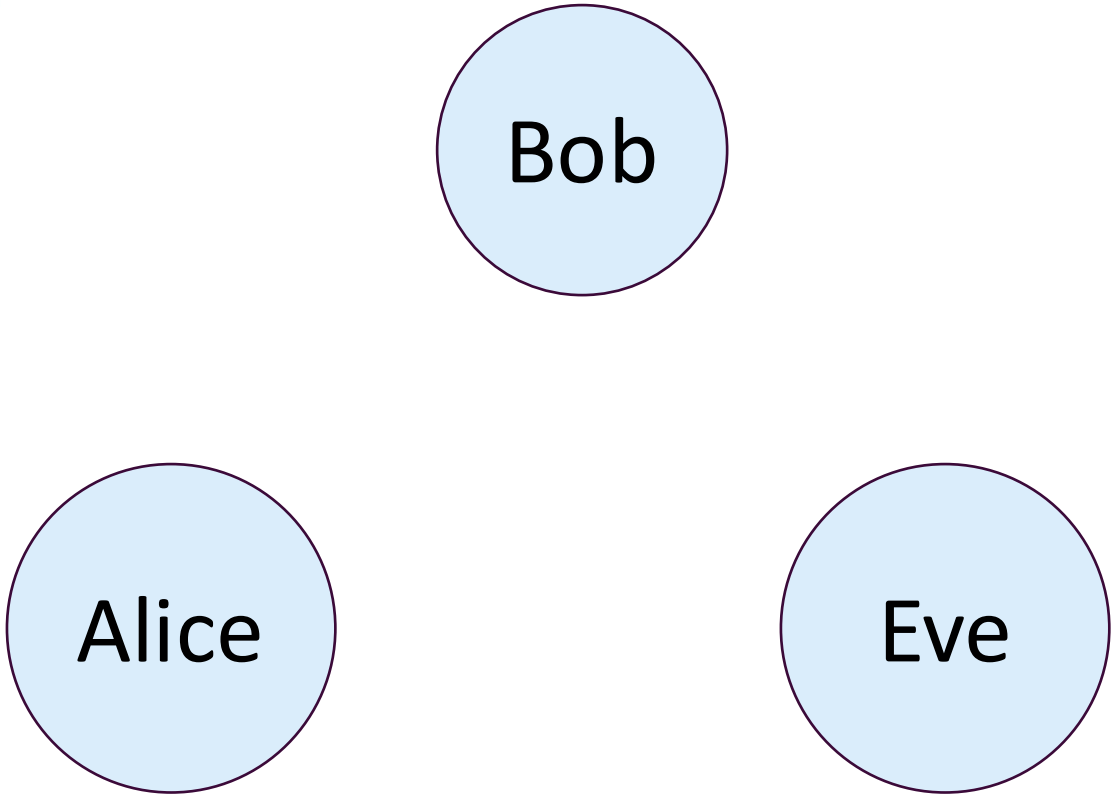
~ 128 billion hyperlinks

~ 2 billion follows



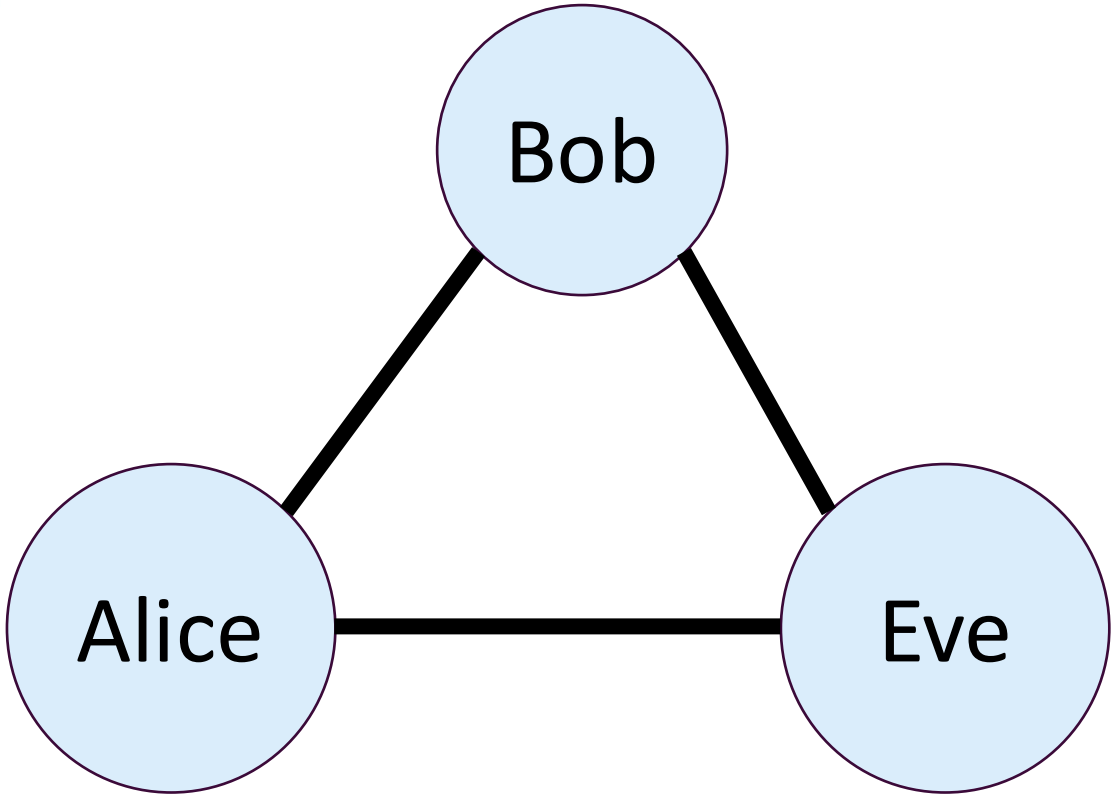
~ 300 million neurons

What is a graph?



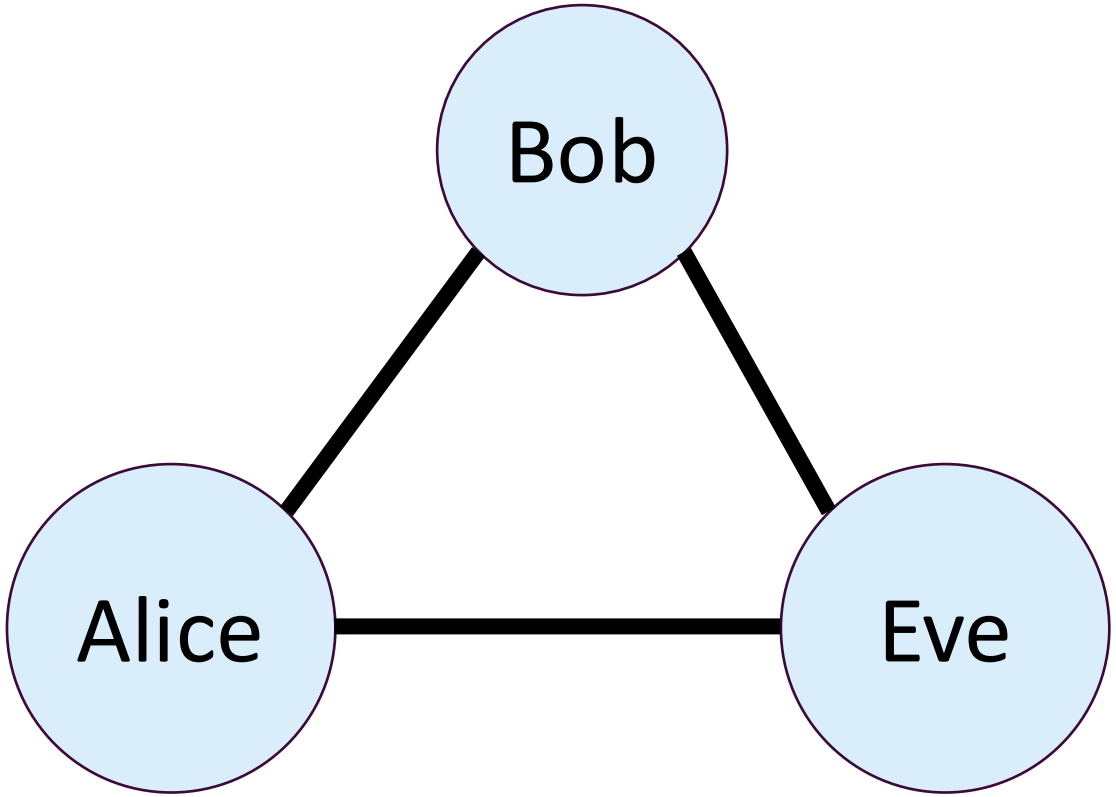
- Graphs consist of **vertices**

What is a graph?



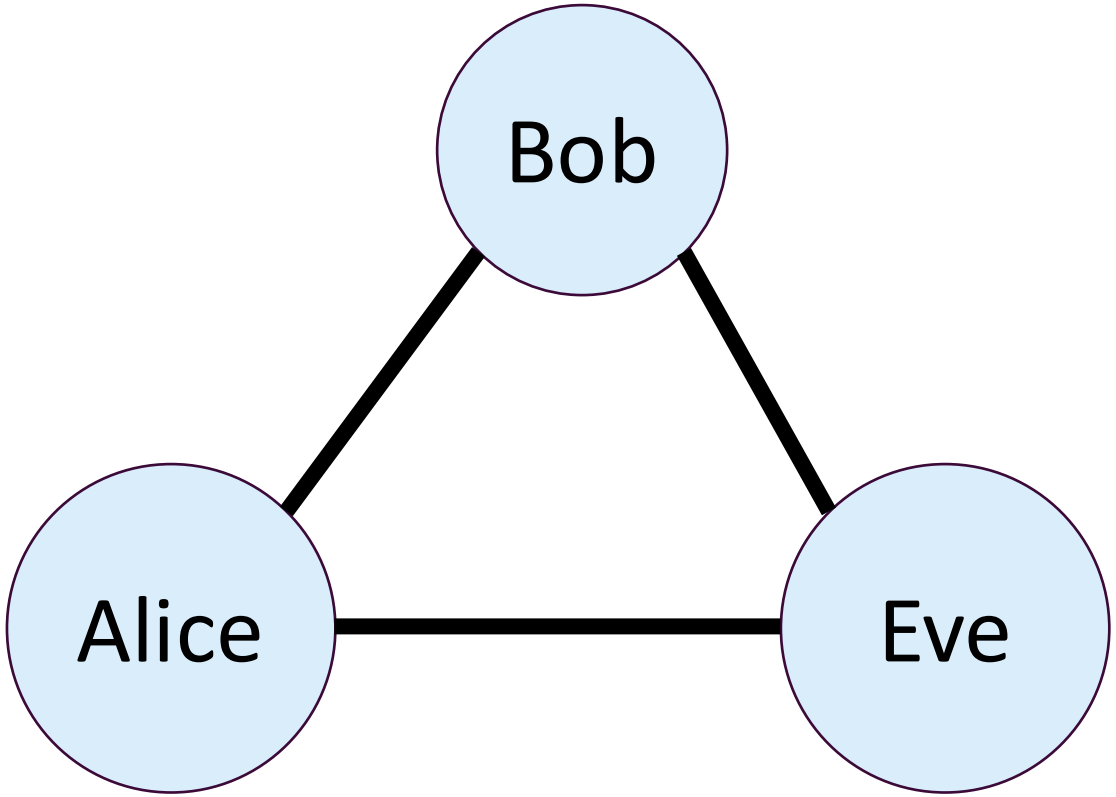
- Graphs consist of **vertices** and **edges**

What is a graph?



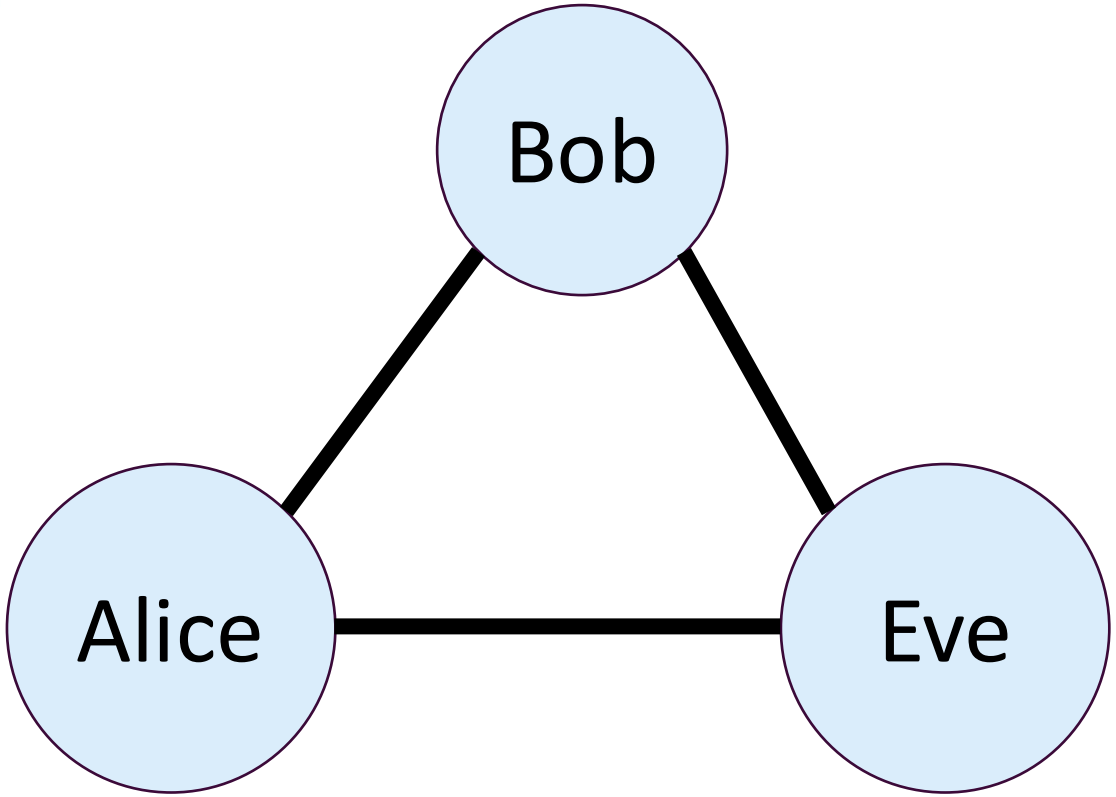
- Graphs consist of **vertices** and **edges**
- **Vertices** represent entities

What is a graph?



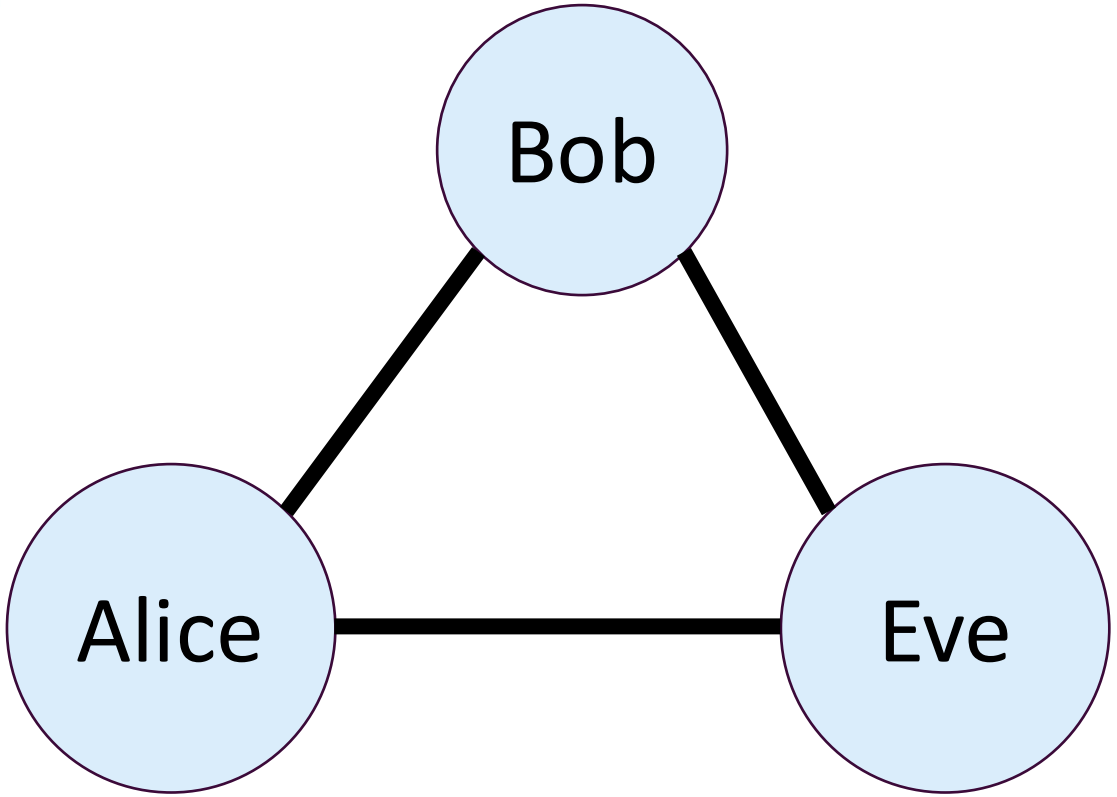
- Graphs consist of **vertices** and **edges**
- **Vertices** represent entities
- **Edges** represent relationships between entities

What is a graph?



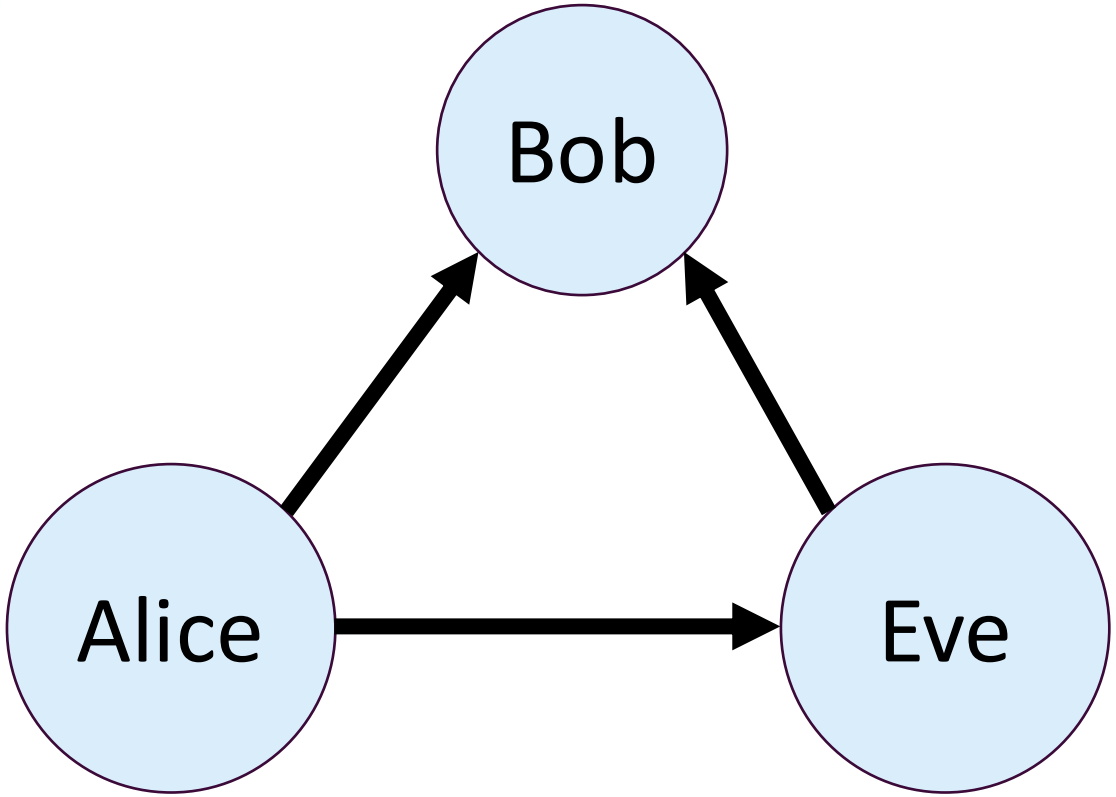
- Graphs consist of **vertices** and **edges**
- **Vertices** represent entities
- **Edges** represent relationships between entities
- Edges can be **undirected**

What is a graph?



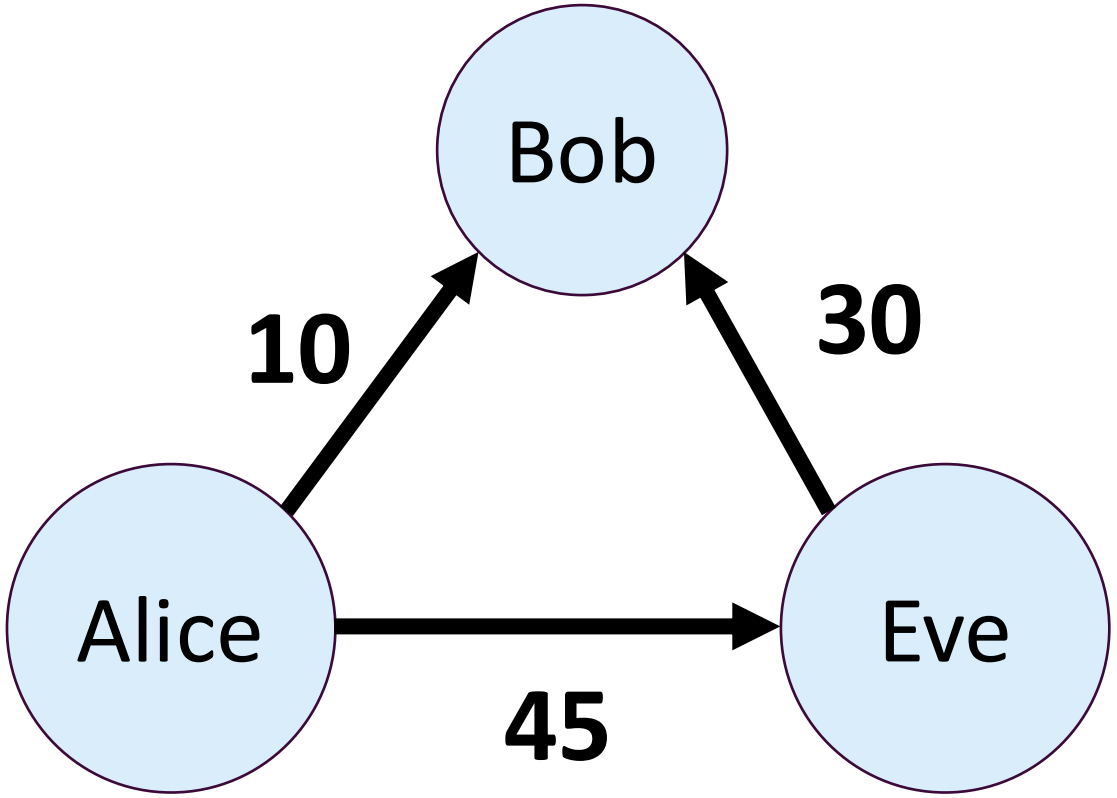
- Graphs consist of **vertices** and **edges**
- **Vertices** represent entities
- **Edges** represent relationships between entities
- Edges can be **undirected** or **directed**

What is a graph?



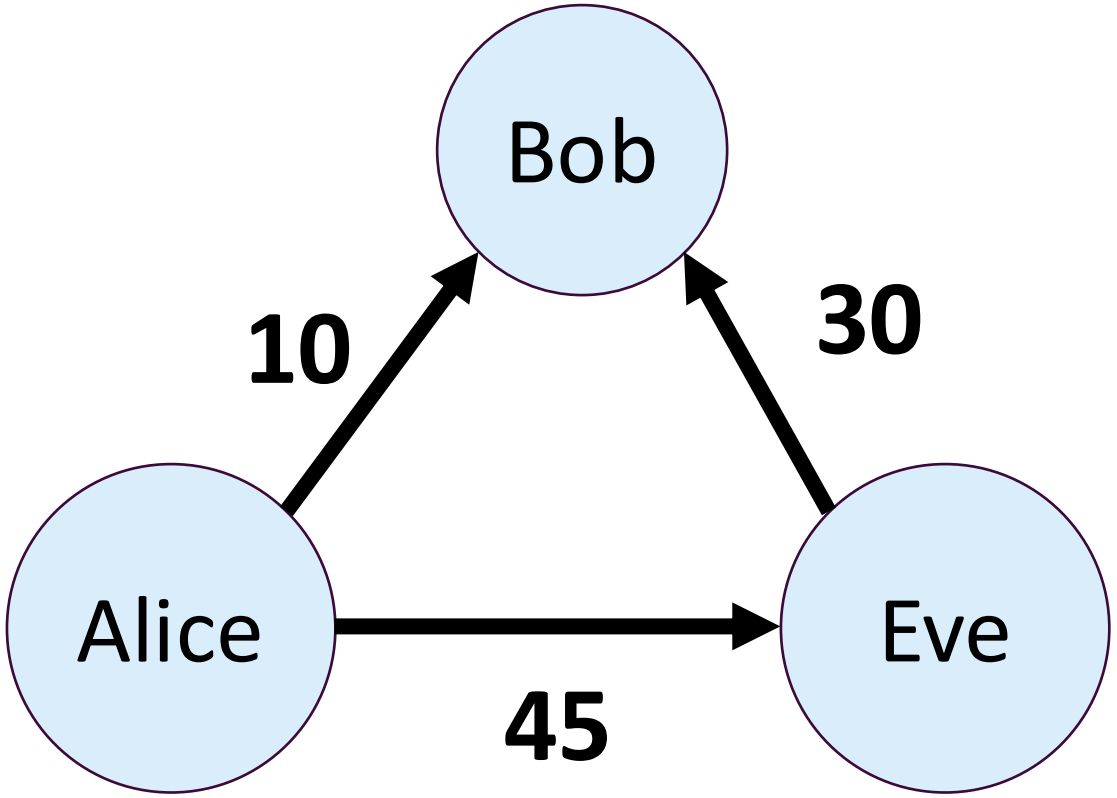
- Graphs consist of **vertices** and **edges**
- **Vertices** represent entities
- **Edges** represent relationships between entities
- Edges can be **undirected** or **directed**

What is a graph?



- Graphs consist of **vertices** and **edges**
- **Vertices** represent entities
- **Edges** represent relationships between entities
- Edges can be **undirected** or **directed**
- Edges can also contain **weights**

What is a graph?



- Graphs consist of **vertices** and **edges**
- **Vertices** represent entities
- **Edges** represent relationships between entities
- Edges can be **undirected** or **directed**
- Edges can also contain **weights**
- Can contain **metadata**

Types of Graphs and Queries

- Example social network queries:

Types of Graphs and Queries

- Example social network queries:
  - **Find all your friends who went to the same high school**

Types of Graphs and Queries

- Example social network queries:
  - **Find all your friends who went to the same high school**
  - **Find all common friends with someone**

Types of Graphs and Queries

- Example social network queries:
  - **Find all your friends who went to the same high school**
  - **Find all common friends with someone**
  - **Recommending friends whom you might know**

You

Types of Graphs and Queries

- Example social network queries:
  - **Find all your friends who went to the same high school**
  - **Find all common friends with someone**
  - **Recommending friends whom you might know**
  - **Advertisement recommendations**

Types of Graphs and Queries

- Transportation network queries:
  - **Find the cheapest way to travel from one city to another**

Types of Graphs and Queries

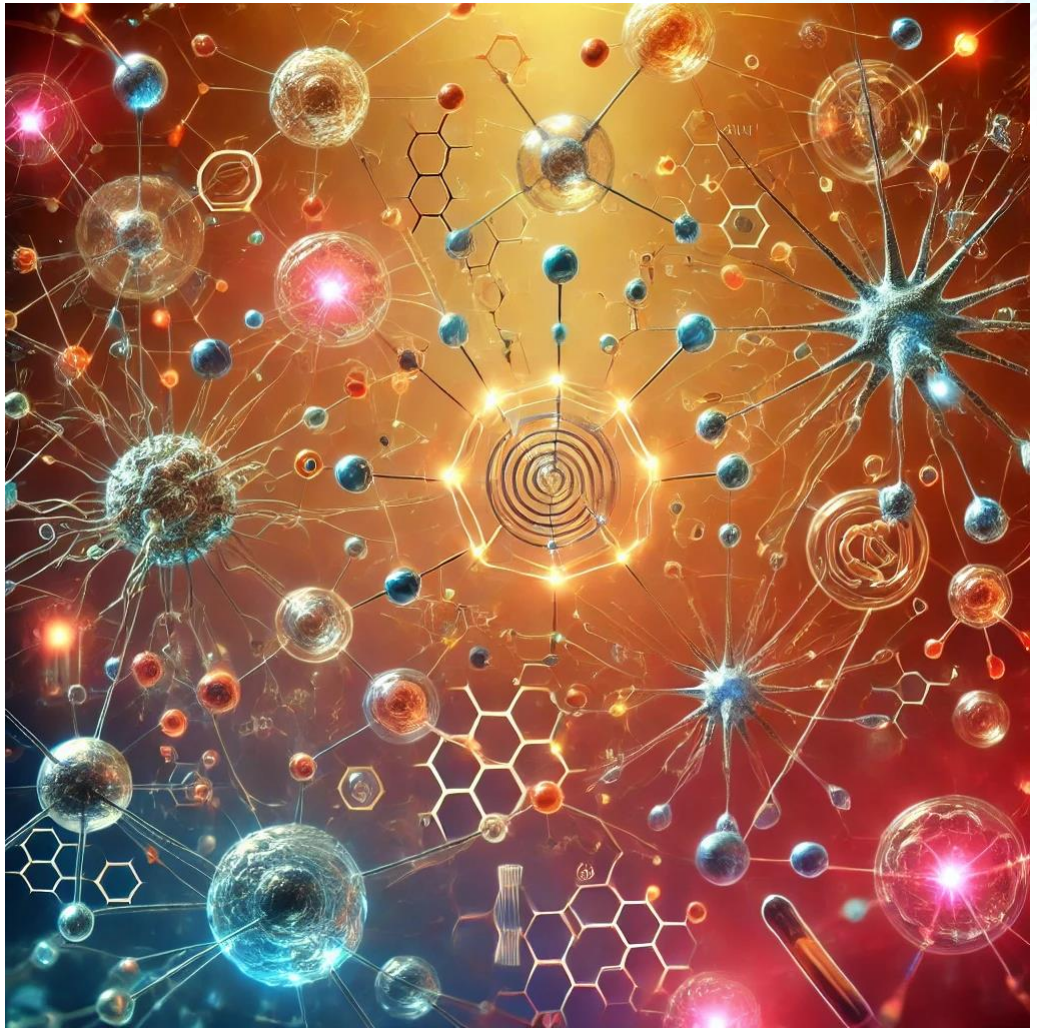
- Transportation network queries:
  - **Find the cheapest way to travel from one city to another**
  - **Decide where to build a hub for profit**

Types of Graphs and Queries

- Transportation network queries:
  - **Find the cheapest way to travel from one city to another**
  - **Decide where to build a hub for profit**
  - **Find the shortest way to visit a set of locations**

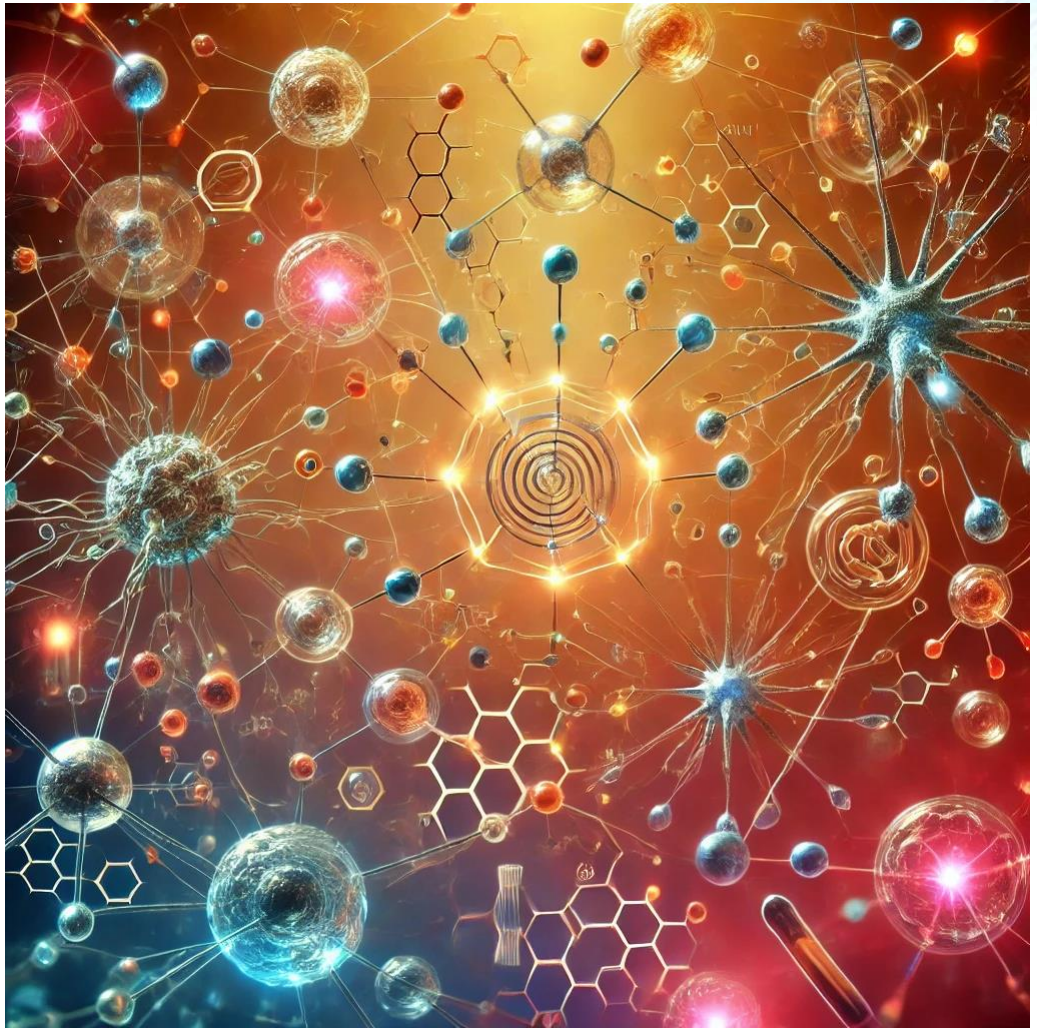
Types of Graphs and Queries

- Biological network queries:
  - **Find patterns in biological networks**



Types of Graphs and Queries

- Biological network queries:
  - **Find patterns in biological networks**
  - **Find similarities between different species**



Graph Representations

- Because graphs are huge in today's real-world environment, we must be able to represent them space-efficiently

Graph Representations

- Because graphs are huge in today's real-world environment, we must be able to represent them space-efficiently
- There are several ways to represent graphs in memory

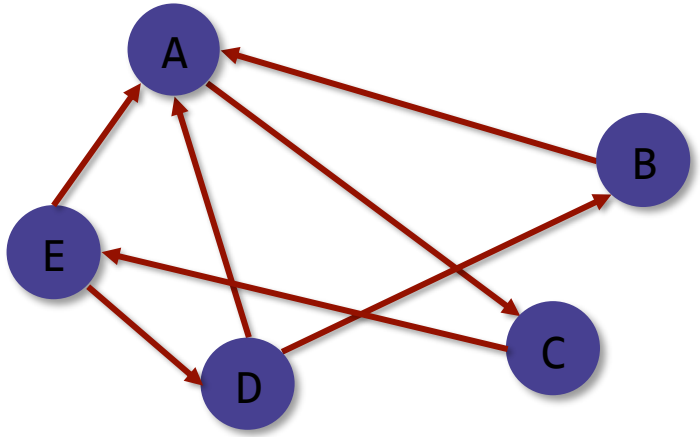
Graph Representations

- Because graphs are huge in today's real-world environment, we must be able to represent them space-efficiently
- There are several ways to represent graphs in memory
- What are the ways to represent data in memory that you've learned previously from your other courses?

Representing A Graph

- Use an **adjacency matrix**
  - (x, y) has a value if there is an edge from Node X to Node Y
  - Value can be the edge value or 1 if edges have no values
- Zeros elsewhere → Sparse Matrix
- Undirected Graph → Symmetric Matrix

from \ to	A	B	C	D	E
A					
B					
C					
D					
E					



Slide source: Courtesy of the 6.106 Lecturers 2022

Graph Scaling

- **Connections don’t scale as fast as the elements**

Slide source: Courtesy of the 6.106 Lecturers 2022

Graph Scaling

- **Connections don’t scale as fast as the elements**
- Example: Friendship Graph
  - In the 424 staff, all of us know each other → $\text{nodes}^2 \approx \text{edges}$

Slide source: Courtesy of the 6.106 Lecturers 2022

Graph Scaling

- **Connections don’t scale as fast as the elements**
- Example: Friendship Graph
  - In the 424 staff, all of us know each other → nodes^2 ≈ edges
  - In the 424 class, many of you knows each other → nodes^2 > edges

Slide source: Courtesy of the 6.106 Lecturers 2022

Graph Scaling

- **Connections don't scale as fast as the elements**
- Example: Friendship Graph
  - In the 424 staff, all of us know each other → $\text{nodes}^2 \approx \text{edges}$
  - In the 424 class, many of you knows each other → $\text{nodes}^2 > \text{edges}$
  - In the world, number of friends is a small constant → $\text{nodes}^2 \gg \text{edges}$

Slide source: Courtesy of the 6.106 Lecturers 2022

Graph Scaling

- **Connections don't scale as fast as the elements**
- Example: Friendship Graph
  - In the 424 staff, all of us know each other → $\text{nodes}^2 \approx \text{edges}$
  - In the 424 class, many of you knows each other → $\text{nodes}^2 > \text{edges}$
  - In the world, number of friends is a small constant → $\text{nodes}^2 \gg \text{edges}$
  - Going from the staff, class, to Yale, to the county, to the world nodes increase fast, but edges increase slowly → sparsity increases

Slide source: Courtesy of the 6.106 Lecturers 2022

Graph Scaling

- **Connections don't scale as fast as the elements**
- Example: Friendship Graph
  - In the 424 staff, all of us know each other → $\text{nodes}^2 \approx \text{edges}$
  - In the 424 class, many of you knows each other → $\text{nodes}^2 > \text{edges}$
  - In the world, number of friends is a small constant → $\text{nodes}^2 \gg \text{edges}$
  - Going from the staff, class, to Yale, to the county, to the world nodes increase fast, but edges increase slowly → sparsity increases

- Most natural and artificial graphs have this property
  - Bigger the graph, higher the sparsity

Slide source: Courtesy of the 6.106 Lecturers 2022

Representations of Data

How do we represent data in memory to optimize performance?

Representing Data

Scalar

0-tensor

Slide source: Courtesy of the 6.106 Lecturers 2022

Representing Data

Scalar

Vector

0-tensor

1-tensor

Slide source: Courtesy of the 6.106 Lecturers 2022

Representing Data

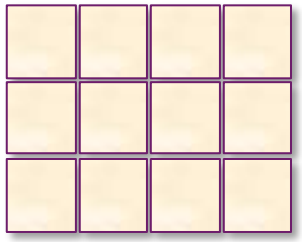
Scalar

0-tensor

Vector

1-tensor

Matrix



2-tensor

Slide source: Courtesy of the 6.106 Lecturers 2022

Representing Data

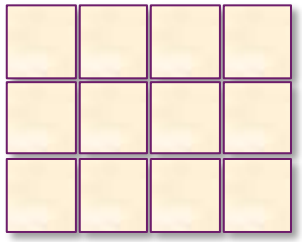
Scalar

0-tensor

Vector

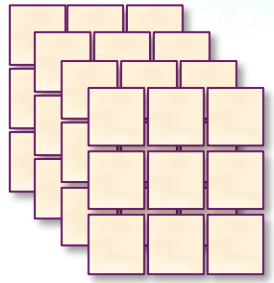
1-tensor

Matrix



2-tensor

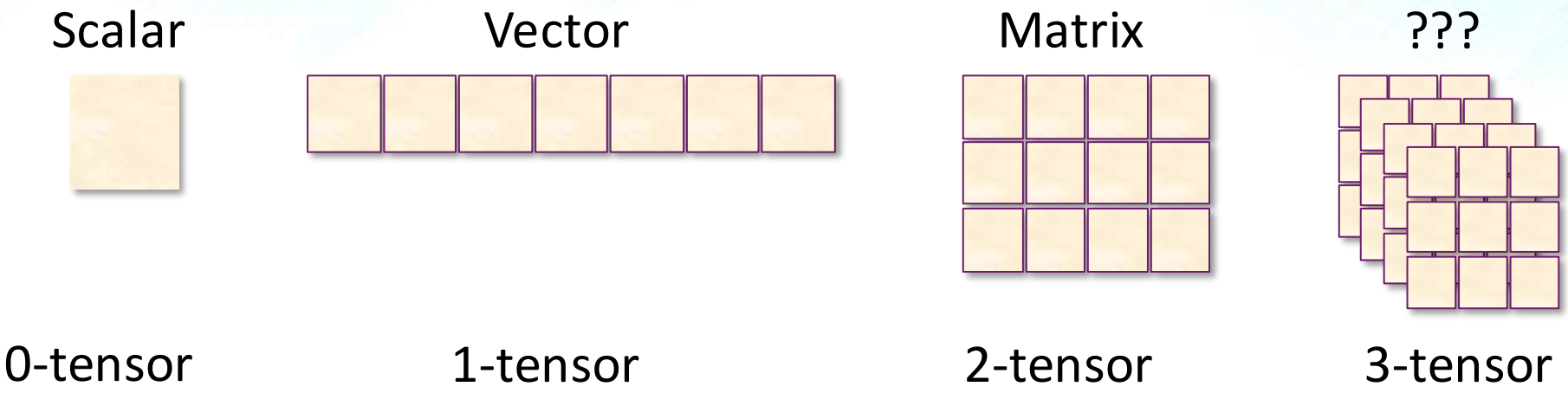
???



3-tensor

Slide source: Courtesy of the 6.106 Lecturers 2022

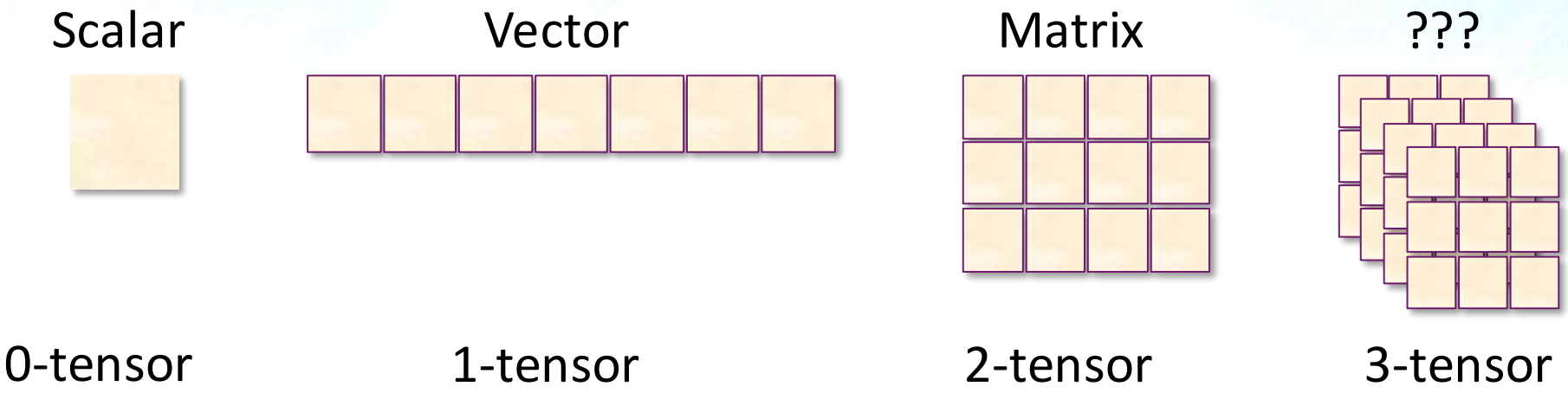
Representing Data



- Mathematicians refer to these as **tensors**
  - In Programming Languages they are **arrays**

Slide source: Courtesy of the 6.106 Lecturers 2022

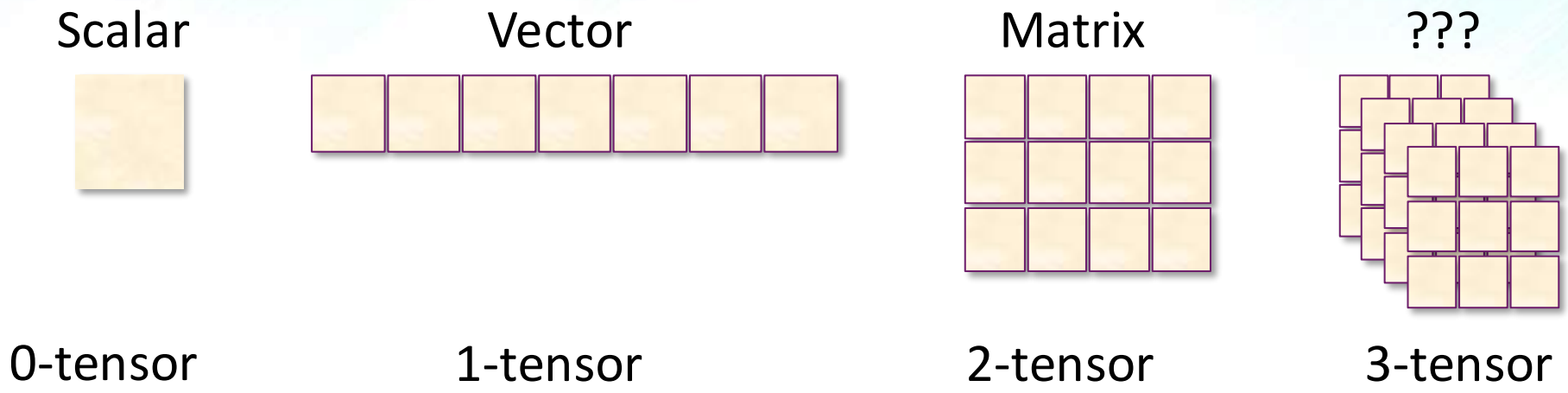
Representing Data



- Mathematicians refer to these as **tensors**
  - In Programming Languages they are **arrays**
- Advantage of Tensors/arrays

Slide source: Courtesy of the 6.106 Lecturers 2022

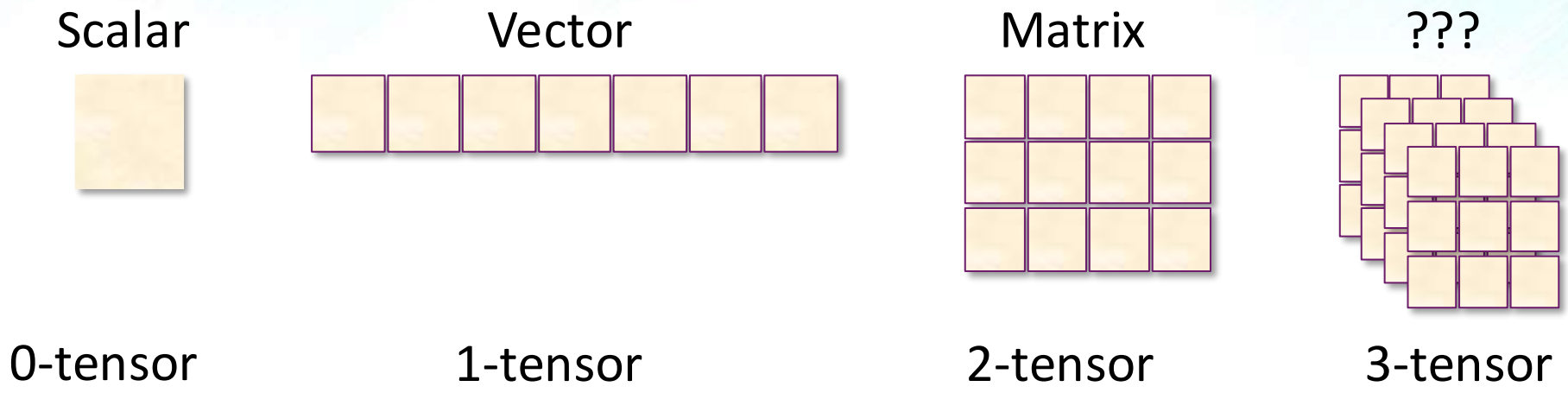
Representing Data



- Mathematicians refer to these as **tensors**
  - In Programming Languages they are **arrays**
- Advantage of Tensors/arrays
  - Simple to understand, use and debug

Slide source: Courtesy of the 6.106 Lecturers 2022

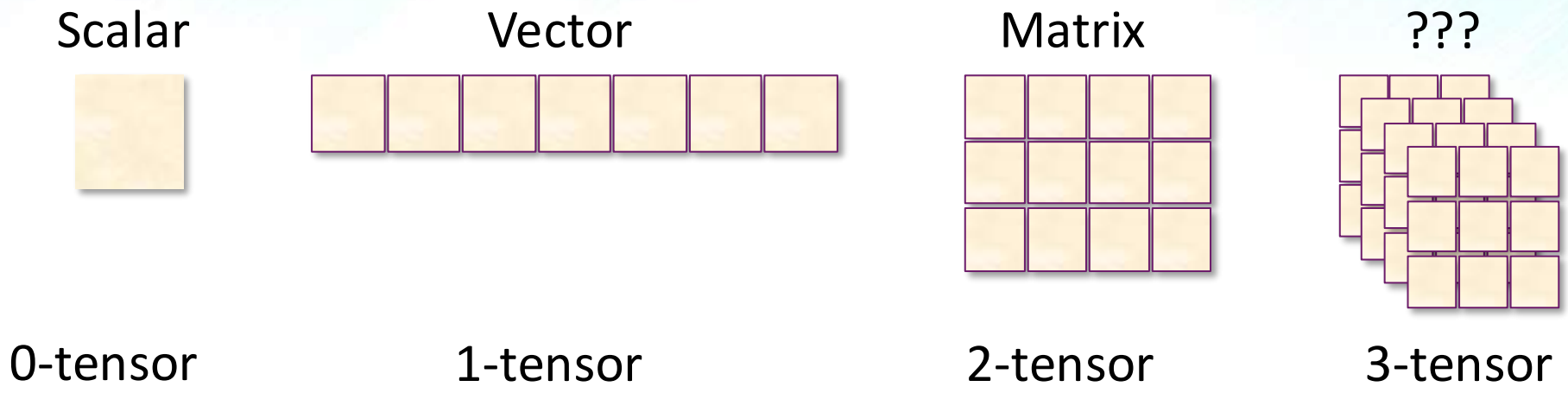
Representing Data



- Mathematicians refer to these as **tensors**
  - In Programming Languages they are **arrays**
- Advantage of Tensors/arrays
  - Simple to understand, use and debug
  - No meta-data needed

Slide source: Courtesy of the 6.106 Lecturers 2022

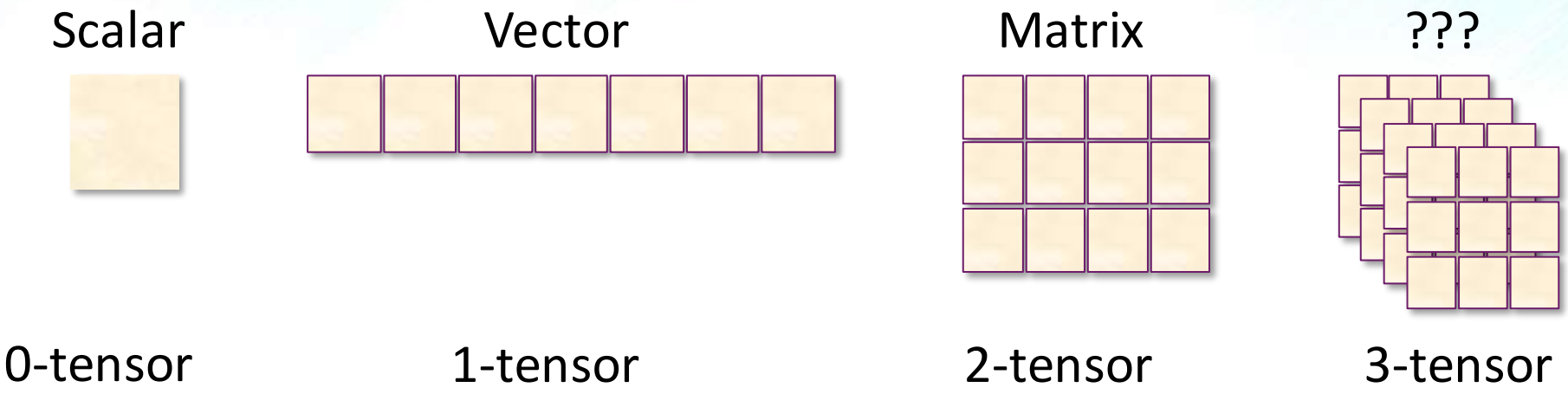
Representing Data



- Mathematicians refer to these as **tensors**
  - In Programming Languages they are **arrays**
- Advantage of Tensors/arrays
  - Simple to understand, use and debug
  - No meta-data needed
  - No pointer chasing

Slide source: Courtesy of the 6.106 Lecturers 2022

Representing Data



- Mathematicians refer to these as **tensors**
  - In Programming Languages they are **arrays**
- Advantage of Tensors/arrays
  - Simple to understand, use and debug
  - No meta-data needed
  - No pointer chasing
  - Works well with caches and prefetchers

Slide source: Courtesy of the 6.106 Lecturers 2022

Arrays/Tensors in FORTRAN

- The first data structure in a programming language

```
      PROGRAM DEMO
      IMPLICIT REAL(A-H,O-Z)
      IMPLICIT INTEGER*3(I-N)

      FUNCTION MXV(A, X)
      DIMENSION A(1000, 500), X(1000)

      DIMENSION Y(500)

      DO 100 I = 1, 500
         Y(I) = 0.0
         DO 200 J = 1, 1000
            Y(I) = Y(I) + A(J, I) * X(J)
200      CONTINUE
100   CONTINUE

      MXV = Y
      RETURN
```

Slide source: Courtesy of the 6.106 Lecturers 2022

Arrays/Tensors in FORTRAN

- The first data structure in a programming language
  - FORTRAN, introduced in 1953 by John Backus, had arrays

```
      PROGRAM DEMO
      IMPLICIT REAL(A-H,O-Z)
      IMPLICIT INTEGER*3(I-N)

      FUNCTION MXV(A, X)
      DIMENSION A(1000, 500), X(1000)

      DIMENSION Y(500)

      DO 100 I = 1, 500
         Y(I) = 0.0
         DO 200 J = 1, 1000
            Y(I) = Y(I) + A(J, I) * X(J)
200      CONTINUE
100   CONTINUE

      MXV = Y
      RETURN
```

Slide source: Courtesy of the 6.106 Lecturers 2022

Arrays/Tensors in FORTRAN

- The first data structure in a programming language
  - FORTRAN, introduced in 1953 by John Backus, had arrays
  - Only compound data type in FORTRAN was arrays

```
      PROGRAM DEMO
      IMPLICIT REAL(A-H,O-Z)
      IMPLICIT INTEGER*3(I-N)

      FUNCTION MXV(A, X)
      DIMENSION A(1000, 500), X(1000)

      DIMENSION Y(500)

      DO 100 I = 1, 500
         Y(I) = 0.0
         DO 200 J = 1, 1000
            Y(I) = Y(I) + A(J, I) * X(J)
200      CONTINUE
100   CONTINUE

      MXV = Y
      RETURN
```

Slide source: Courtesy of the 6.106 Lecturers 2022

Arrays/Tensors in FORTRAN

- The first data structure in a programming language
  - FORTRAN, introduced in 1953 by John Backus, had arrays
  - Only compound data type in FORTRAN was arrays
- All subsequent languages have arrays and loops

```
      PROGRAM DEMO
      IMPLICIT REAL(A-H,O-Z)
      IMPLICIT INTEGER*3(I-N)

      FUNCTION MXV(A, X)
      DIMENSION A(1000, 500), X(1000)

      DIMENSION Y(500)

      DO 100 I = 1, 500
         Y(I) = 0.0
         DO 200 J = 1, 1000
            Y(I) = Y(I) + A(J, I) * X(J)
200      CONTINUE
100   CONTINUE

      MXV = Y
      RETURN
```

Slide source: Courtesy of the 6.106 Lecturers 2022

Structured Data

- Normally large data sets

Slide source: Courtesy of the 6.106 Lecturers 2022

Structured Data

- Normally large data sets
  - Come from nature (from sensors)

Slide source: Courtesy of the 6.106 Lecturers 2022

Structured Data

- Normally large data sets
  - Come from nature (from sensors)
  - Human-generated (by computers)

Slide source: Courtesy of the 6.106 Lecturers 2022

Structured Data

- Normally large data sets
  - Come from nature (from sensors)
  - Human-generated (by computers)
- Most data have some underling structure

Slide source: Courtesy of the 6.106 Lecturers 2022

Structured Data

- Normally large data sets
  - Come from nature (from sensors)
  - Human-generated (by computers)
- Most data have some underling structure
  - Can we take advantage of the structure?

Slide source: Courtesy of the 6.106 Lecturers 2022

Structured Data

- Normally large data sets
  - Come from nature (from sensors)
  - Human-generated (by computers)
- Most data have some underling structure
  - Can we take advantage of the structure?
- What are the types of structure?

Slide source: Courtesy of the 6.106 Lecturers 2022

Structured Data

- Normally large data sets
  - Come from nature (from sensors)
  - Human-generated (by computers)
- Most data have some underling structure
  - Can we take advantage of the structure?
- What are the types of structure?
  - Repeated data values

Slide source: Courtesy of the 6.106 Lecturers 2022

Structured Data

- Normally large data sets
  - Come from nature (from sensors)
  - Human-generated (by computers)
- Most data have some underling structure
  - Can we take advantage of the structure?
- What are the types of structure?
  - Repeated data values
  - Sparse data values

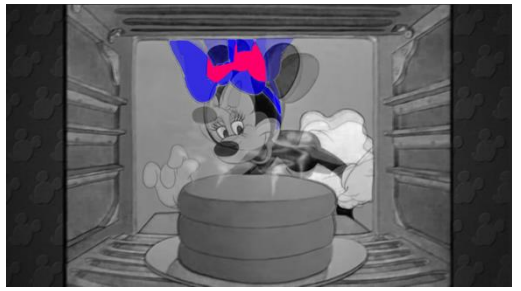
Slide source: Courtesy of the 6.106 Lecturers 2022

Structured Data

- Normally large data sets
  - Come from nature (from sensors)
  - Human-generated (by computers)
- Most data have some underling structure
  - Can we take advantage of the structure?
- What are the types of structure?
  - Repeated data values
  - Sparse data values
  - Symmetric data values

Slide source: Courtesy of the 6.106 Lecturers 2022

Repeated Data Values



- **Each pixel on each frame → a lot of data**
- In each fame, repeated data
- Data repeated across frames
- Solution: Data Compression
  - Lossy and Lossless compression

Slide source: Courtesy of the 6.106 Lecturers 2022

Sparse Data Values

Many real-world data sets are **sparse**
Large number of **values are zero**

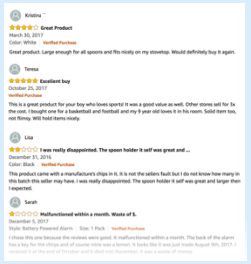
Slide source: Courtesy of the 6.106 Lecturers 2022

Sparse Data Values

Data Analytics

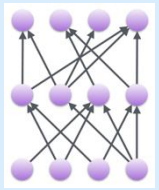
Movies

Social Networks

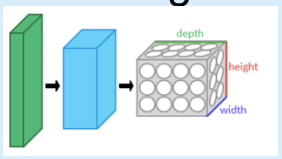


Product Reviews

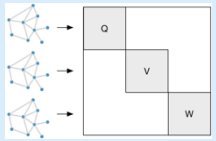
Machine Learning



Sparse Networks

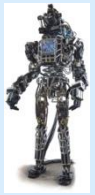


Sparse Convolutional Networks

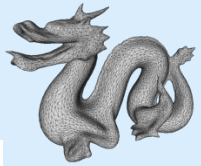


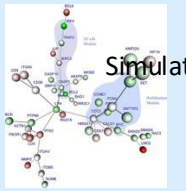
Graph Convolutional Network

Science and Engineering



Robotics

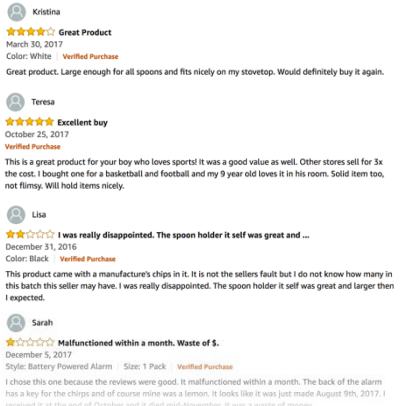


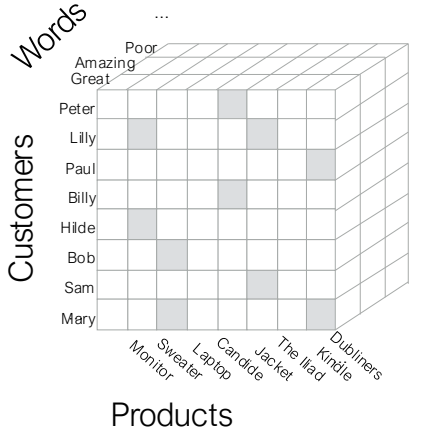


Simulations

Computational Biology

amazon

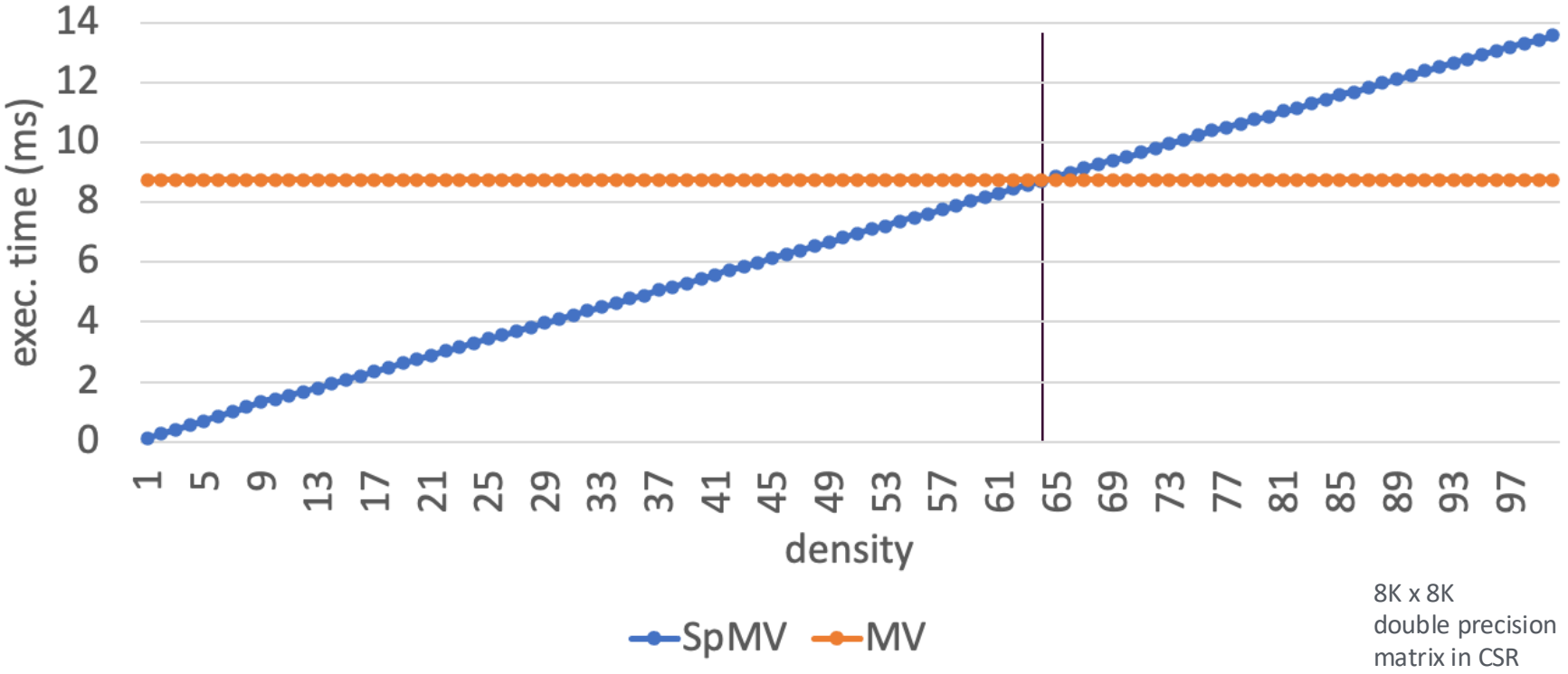




Extremely sparse
Dense storage: 107 Exabytes
Sparse storage: 13 Gigabytes

Slide source: Courtesy of the 6.106 Lecturers 2022

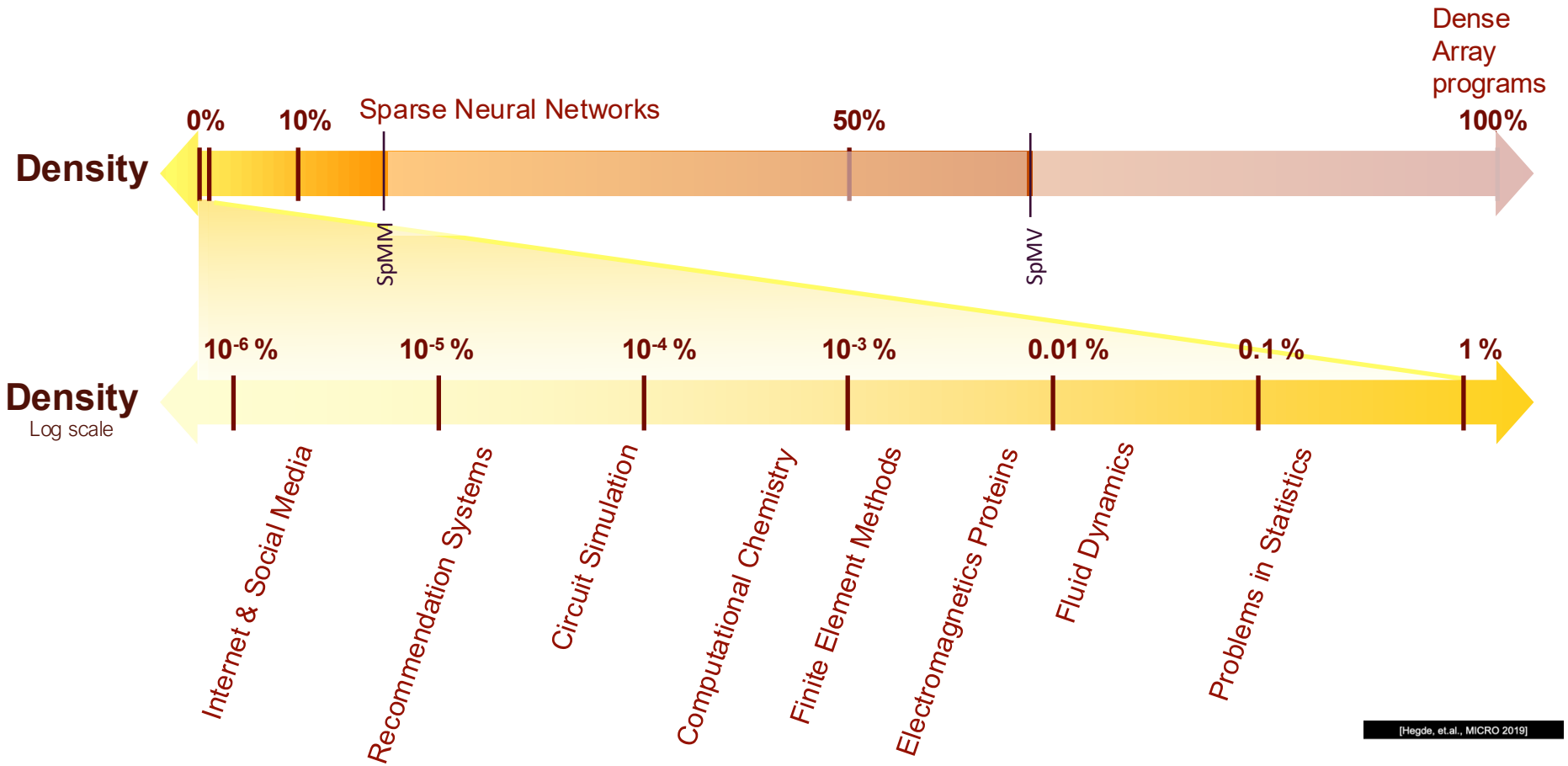
Ignoring Sparsity is Throwing Away Performance



Sparse Matrix Vector Multiplication (SpMV)

Slide source: Courtesy of the 6.106 Lecturers 2022

Sparsity is **EVERYWHERE!**



Slide source: Courtesy of the 6.106 Lecturers 2022

Graphs are sparse!

- **Few Edges Relative to Nodes**:

Graphs are sparse!

- **Few Edges Relative to Nodes**:
  - If a graph has *N* nodes, a **dense graph** would have around **N^2 edges**

Graphs are sparse!

- **Few Edges Relative to Nodes**:
  - If a graph has *N* nodes, a **dense graph** would have around **N^2 edges**
  - Sparse graph has **far fewer edges** (often proportional to *N* or *N log N*)

Graphs are sparse!

- **Few Edges Relative to Nodes**:
  - If a graph has *N* nodes, a **dense graph** would have around **N^2 edges**
  - Sparse graph has **far fewer edges** (often proportional to *N* or *N log N*)
- **Adjacency Matrix is Mostly Zeros**:

Graphs are sparse!

- **Few Edges Relative to Nodes**:
  - If a graph has *N* nodes, a **dense graph** would have around **N^2 edges**
  - Sparse graph has **far fewer edges** (often proportional to *N* or *N log N*)
- **Adjacency Matrix is Mostly Zeros**:
  - A **dense graph** has a nearly **full adjacency matrix**

Graphs are sparse!

- **Few Edges Relative to Nodes**:
  - If a graph has *N* nodes, a **dense graph** would have around **N^2 edges**
  - Sparse graph has **far fewer edges** (often proportional to *N* or *N log N*)
- **Adjacency Matrix is Mostly Zeros**:
  - A **dense graph** has a nearly **full adjacency matrix**
  - A sparse graph's **matrix is predominantly zeros**

Graphs are sparse!

- **Few Edges Relative to Nodes**:
  - If a graph has *N* nodes, a **dense graph** would have around **N^2 edges**
  - Sparse graph has **far fewer edges** (often proportional to *N* or *N log N*)
- **Adjacency Matrix is Mostly Zeros**:
  - A **dense graph** has a nearly **full adjacency matrix**
  - A sparse graph's **matrix is predominantly zeros**
- **Efficient Storage & Processing**:

Graphs are sparse!

- **Few Edges Relative to Nodes**:
  - If a graph has *N* nodes, a **dense graph** would have around **N^2 edges**
  - Sparse graph has **far fewer edges** (often proportional to *N* or *N log N*)
- **Adjacency Matrix is Mostly Zeros**:
  - A **dense graph** has a nearly **full adjacency matrix**
  - A sparse graph's **matrix is predominantly zeros**
- **Efficient Storage & Processing**:
  - Sparse graphs require **specialized data structures** like compressed sparse row (CSR) format

Most real-world graphs are sparse!

- **Social Networks** (e.g., Twitter, Facebook)
  - Most users have a few connections

Most real-world graphs are sparse!

- **Social Networks** (e.g., Twitter, Facebook)
  - Most users have a few connections
- **Web Graphs**
  - Websites link to a **small subset** of other sites, not all pages

Most real-world graphs are sparse!

- **Social Networks** (e.g., Twitter, Facebook)
  - Most users have a few connections
- **Web Graphs**
  - Websites link to a **small subset** of other sites, not all pages
- **Recommendation Systems**
  - User-item interactions form a **sparse** bipartite graph

Most real-world graphs are sparse!

- **Social Networks** (e.g., Twitter, Facebook)
  - Most users have a few connections
- **Web Graphs**
  - Websites link to a **small subset** of other sites, not all pages
- **Recommendation Systems**
  - User-item interactions form a **sparse** bipartite graph
- **Biological Networks**
  - Gene or protein interaction networks are sparse

Most real-world graphs are sparse!

- **Social Networks** (e.g., Twitter, Facebook)
  - Most users have a few connections
- **Web Graphs**
  - Websites link to a **small subset** of other sites, not all pages
- **Recommendation Systems**
  - User-item interactions form a **sparse** bipartite graph
- **Biological Networks**
  - Gene or protein interaction networks are sparse
- **Road Networks**
  - Cities are connected via roads, but not every location connects to every other location

Storing Sparse Data

Including many graphs! Graphs are sparse!

Data Representations in CSR

- **Compressed Sparse Row (CSR)** format:
  - One of the most widely used **sparse matrix storage formats**

Data Representations in CSR

- **Compressed Sparse Row (CSR)** format:
  - One of the most widely used **sparse matrix storage formats**
  - Enables efficient **storage, computation, and parallelization**

Data Representations in CSR

- **Compressed Sparse Row (CSR)** format:
  - One of the most widely used **sparse matrix storage formats**
  - Enables efficient **storage, computation, and parallelization**
- CSR is a **memory-efficient format** for storing sparse matrices, where most of the elements are **zeroes**

Data Representations in CSR

- **Compressed Sparse Row (CSR)** format:
  - One of the most widely used **sparse matrix storage formats**
  - Enables efficient **storage, computation, and parallelization**
- CSR is a **memory-efficient format** for storing sparse matrices, where most of the elements are **zeroes**
- **Saves memory** by storing only nonzero values

Data Representations in CSR

- **Compressed Sparse Row (CSR)** format:
  - One of the most widely used **sparse matrix storage formats**
  - Enables efficient **storage, computation, and parallelization**
- CSR is a **memory-efficient format** for storing sparse matrices, where most of the elements are **zeroes**
- **Saves memory** by storing only nonzero values
- **Speeds up operations** like matrix-vector multiplication (SpMV)

Data Representations in CSR

- **Compressed Sparse Row (CSR)** format:
  - One of the most widely used **sparse matrix storage formats**
  - Enables efficient **storage, computation, and parallelization**
- CSR is a **memory-efficient format** for storing sparse matrices, where most of the elements are **zeroes**
- **Saves memory** by storing only nonzero values
- **Speeds up operations** like matrix-vector multiplication (SpMV)
- **Supports parallelization** efficiently due to predictable memory access patterns

What is CSR?

- CSR represents a sparse matrix using **three arrays**:

What is CSR?

- CSR represents a sparse matrix using **three arrays**:
  - **Values (VAL):** Stores all nonzero elements row-wise.

What is CSR?

- CSR represents a sparse matrix using **three arrays**:
  - **Values (VAL):** Stores all nonzero elements row-wise.
  - **Column Indices (COL):** Stores the column indices corresponding to each nonzero element.

What is CSR?

- CSR represents a sparse matrix using **three arrays**:
  - **Values (VAL):** Stores all nonzero elements row-wise.
  - **Column Indices (COL):** Stores the column indices corresponding to each nonzero element.
  - **Row Pointers (ROW_PTR):** Marks the **starting index** of each row in the VAL and COL arrays.

Example CSR Format

- Consider the following **4×5 sparse matrix**:

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Consider the following **4×5 sparse matrix**:

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

- Let’s convert to CSR step by step

Example CSR Format

- Create the VAL array

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the VAL array
- Store all **nonzero elements** row by row:

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the VAL array
- Store all **nonzero elements** row by row:

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the COL (Column Indices) Array

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the COL (Column Indices) Array
- For each nonzero element in VAL, store its **column index**:

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the COL (Column Indices) Array
- For each nonzero element in VAL, store its **column index**:

COL=[2,4,0,3,1,4]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the COL (Column Indices) Array
- For each nonzero element in VAL, store its **column index**:

COL=[2,4,0,3,1,4]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

- **3** is in col **2, 4** is in col **4** etc.

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

Example CSR Format

- Create the ROW_PTR (Row Pointers) Array
- This array marks the **start of each row** in VAL and COL:
  - Start at **0** (beginning of VAL)
  - Each row's starting index is recorded, while empty rows repeat the last index

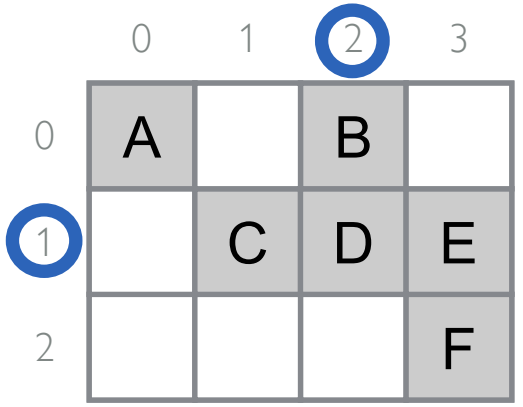
VAL = [3,4,5,7,8,6]

$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 7 & 0 \\ 0 & 8 & 0 & 0 & 6 \end{bmatrix}$$

ROW_PTR = [0,2,2,4,6]

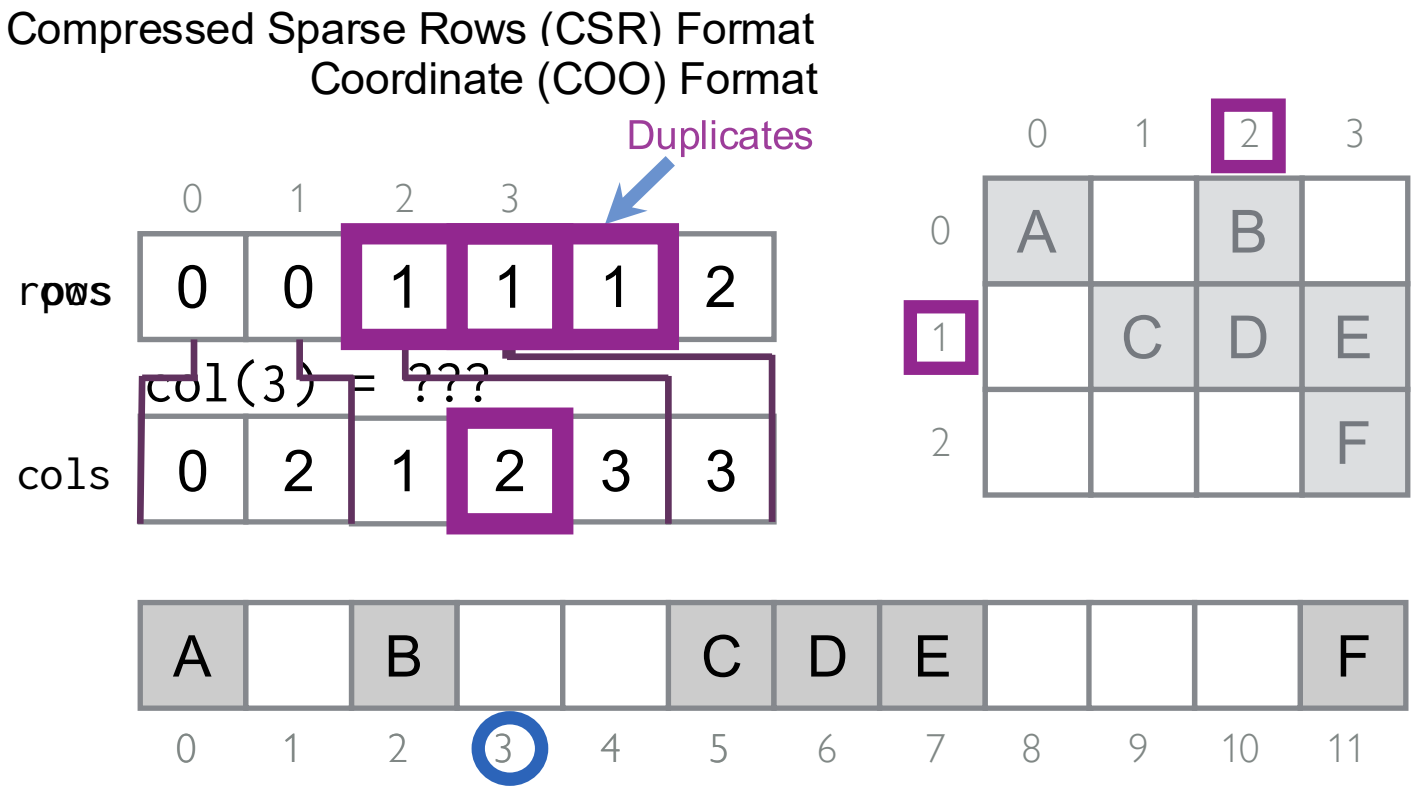
Data Representations in CSR

```
locate(1,2) = 1*4 + 2
            = 6
```



Slide source: Courtesy of the 6.106 Lecturers 2022

Data Representations in CSR



Slide source: Courtesy of the 6.106 Lecturers 2022

Lecture 12 Quiz