Block-Jacobi and other Parallel Algorithms for Sparse Matrices in OCaml

Samuel Sanft
Adviser: Andrew Appel

Abstract

This paper investigates and explains how parallel algorithms for sparse matrices can be implemented in OCaml v5. It describes a research project that involves building an versatile module for constructing and storing sparse matrices in OCaml and implementing efficient and parallel algorithms for vector and matrix operations. It also describes how this module can be used to implement the Jacobi and Block-Jacobi methods for solving linear systems. Lastly it details timing results, showing that significant speed-up can be achieved using multicore parallelism in OCaml v5, but that the effectiveness of parallelization depends on the individual problem. For instance, the Block-Jacobi method sees significantly better speed-up when parallelized compared to the Jacobi method.

1. Introduction

This paper begins by explaining the motivation for studying parallel algorithms for sparse matrices in OCaml. Next it details the goals of the research project, followed by a short overview of related research to the topic. Then it describes the implementation of a sparse matrix module in OCaml, parallel vector, matrix and matrix-vector operations, and the Jacobi and Block-Jacobi methods. Lastly it shows and analyzes the results from timing tests and describes opportunities for future work in this area.

2. Motivation

As the title and introduction of this paper suggest, this research project contains multiple different aspects, each of which have their own justifications for studying.

2.1. Why study parallel algorithms?

The operation speed of sequential computers has been increasing exponentially for decades, however, those increases in speed are now coming at greater and greater costs. In response, researchers and manufacturers have looked to increase the parallel capabilities of computers as well, increasing the number of processors available in both our top end supercomputers, and our everyday devices. However, in order to take advantage of this increasing computing power, there is a need to design algorithms that can run well in parallel [3]. Despite this need, many introductory algorithms courses or textbooks focus primarily, or even exclusively, on sequential implementations of algorithms. This establishes a need for the study of parallel algorithms specifically, as well as their implementation in modern programming languages and their performance on modern devices.

2.2. Why study sparse linear systems?

Sparse, often large, linear systems have applications in numerous fields including structural engineering, computer graphics/vision, economic and financial modeling, mathematics and statistics, and optimization [7]. One example of an algorithm that exploits sparsity is Google's PageRank which was designed by Larry Page and Sergey Brin and was used to rank the relevance of search results in Google's original search engine [4]. The algorithm determines a website's relevance, based on how many other sites link to it, and the relevance of those sites. It uses an adjacency matrix to represent a graph of links between websites, where there may be millions or even billions of total sites that have been indexed, but each one only has links to a handful of others. Matrices like these can be very useful, however, storing them and performing operations on them using traditional methods is incredibly inefficient, as the majority of all entries are zeroes, and thus have no effect on the problem. This requires developing special ways of dealing with sparse matrices, that may be very different from the ways we handle dense matrices.

2.3. Why use OCaml?

There are a number of reasons why I chose OCaml for this project. The first is that while there are a number of packages available in OCaml for scientific computing or linear algebra, none of them offer support for sparse matrices. This project offers an opportunity to contribute something new and valuable to the OCaml scientific computing and numerical linear algebra communities. The second is that multicore programming, or shared-memory parallel programming, in OCaml has only recently received official support with OCaml v5 [10]. With this project I hope to measure the capabilities and effectiveness of this new feature in the language.

3. Goals

There are multiple goals for this research project. The first is to design and implement a simple module for using sparse matrices in OCaml. This module should be functional and versatile enough to be used as a building block in a wide array of applications that require sparse matrices. As mentioned before, no open-source package currently exists for using sparse matrices, so this could be a valuable contribution to the OCaml scientific computing community. The second goal is to demonstrate an application of sparse matrices by implementing the Jacobi and Block-Jacobi methods, using the sparse matrix module. The Jacobi method, and it's variant, the Block Jacobi method, are iterative methods for solving linear systems of the form $Ax = b$, where A is a square matrix containing real values. These methods are suitable for demonstrating the utility of this sparse matrix module, as they can efficiently solve large, sparse, diagonally-dominant systems to arbitrary precision, and are easily parallelizable. Their implementation will require implementing basic operations for vectors and matrices, including taking dot products and vector norms, vector addition and subtraction, matrix-vector multiplication and LU decomposition. All of these operations are easily parallelizable and will be implemented in sequence and in parallel. This will allow for the Jacobi and Block-Jacobi methods to run in both sequence and parallel, so that the effectiveness of multicore programming in OCaml can be measured.

4. Background Work

4.1. Parallelism

Blelloch has written numerous papers about parallel programming and algorithms. He describes a useful framework for measuring the runtime of parallel algorithms in terms of work and span, where work represents the total amount of computing time across all processors, and span represents the amount of time that the algorithm takes to finish as the number of processors increases to infinity. He discusses this as well as other considerations for implementing parallel algorithms in his paper "Programming Parallel Algorithms" [2].

4.1.1. Parallelism in OCaml In OCaml v5, multicore programming is made available via the `Domain` module [8]. The module allows the programmer to spawn one or more Domains, which are assigned a task at creation and complete their assigned task in parallel on a seperate core, before returning the result to the caller. The module has two functions that are relevant to this project. `spawn` has type `(unit -> 'a) -> 'a t`. Calling `spawn f` will return a Domain that will execute `f` and eventually return it's result or raise an exception. `join` has type `'a t -> 'a`. Calling `join` on a Domain will block the programs execution until the Domain has finished executing, and then return it's result. Using these two functions, it is possible to execute one function twice in parallel. The following function will spawn two domains, `domain_a` and `domain_b`, and then wait for and return the results of `domain_a` and `domain_b`.

```
let execute_twice (f : unit -> 'a) : ('a * 'a) =
  let domain_a = Domain.spawn f in
  let domain_b = Domain.spawn f in
  let result_a = Domain.join domain_a in
  let result_b = Domain.join domain_b in
  result_a, result_b
```

It is also possible to spawn an arbitrary amount of Domains (although it is not advised to spawn more domains then there are cores available). The following function will take arguments `f : int -> 'a` and `n : int`, call `f` on all the integers `0, ..., n - 1` in parallel, and return a list of the results.

```
let call_f_n_times (f : int -> 'a) (n : int) : 'a list =
  let init_domain (i : int) = Domain.spawn (fun _ -> f i) in
  let domains = List.init n init_domain in
  List.map Domain.join domains
```

There are additional libraries for multicore and parallel programming in OCaml that are made available through OPAM, OCaml's package manager, that abstract the use of Domains, so that they don't have to be managed directly, however, their use will not be discussed in this paper (see Future Work for more info).

4.2. Sparse Matrices

Davis and Hu have compiled a large collection of sparse matrices known as the SuiteSparse Matrix Collection [7]. This collection contains sparse matrices of various sizes from a number of different real-world sources. Many matrices sourced from this collection will be used for testing and timing purposes in the Results section of this paper.

4.3. Scientific Computing in OCaml

Wang, Zhao, and Mortier discuss different applications of scientific computing in OCaml, including a number of linear algebra operations, in their book [15]. Their book also describes and documents the OWL open source package for OCaml, which offers support for "N-dimensional arrays, linear algebra, regressions, fast Fourier transforms, and many advanced mathematical and statistical functions" [14]. Unfortunately, their book does not contain any work related to sparse matrices and their package only offers support for dense matrices.

4.4. Jacobi and Block-Jacobi Methods

The Jacobi and Block-Jacobi Methods are iterative methods for solving linear systems of equations. Iterative methods are often the preferred method for solving linear systems when the size of the system is large enough to make exact methods, such as by inversion or Gaussian-Elimination, impractical. Barrett et al. discuss the Jacobi and Block-Jacobi methods, as well as many other iterative methods and their implementations for solving linear systems in their book [1], though a

brief overview will be provided here.

The goal of the the Jacobi method is to find an approximate solution to the linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ and $x, b \in \mathbb{R}^n$. The method begins by splitting A into the matrices D, L and U, where D is a diagonal matrix, L is a lower triangular matrix and U is an upper triangular matrix:

$$A = D+L+U, D = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}, L+U = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}$$

Next an arbitrary x_0 is selected. Any value is sufficient, so if an estimate for x is not available, then the 0-vector is often selected. The approximation for x is then updated iteratively using the following calculation.

$$x_{k+1} = D^{-1}(b-(L+U)x_k)$$

There are a number of ways to determine when to stop running the algorithm, but a common stopping condition is to define a parameter $\varepsilon > 0$ that is sufficiently small, and stop when $||x_{k+1} - x_k||^2 < \varepsilon$.

The Block-Jacobi method is very similar, with the only difference being in the way A is partitioned. Instead of defining D as a diagonal matrix, with $L+U$ containing the remaining values of A, we define D as a block-diagonal matrix. For example, if A is divided into $p \times p$ blocks of size $\frac{n}{p} \times \frac{n}{p}$:

$$A = \left[\begin{array}{c|c|c|c} \alpha_{11} & \alpha_{12} & \cdots & \alpha_{1p} \\ \hline \alpha_{21} & \alpha_{22} & \cdots & \alpha_{2p} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline \alpha_{p1} & \alpha_{p2} & \cdots & \alpha_{pp} \end{array}\right], D = \left[\begin{array}{c|c|c|c} \alpha_{11} & 0 & \cdots & 0 \\ \hline 0 & \alpha_{22} & \cdots & 0 \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline 0 & 0 & \cdots & \alpha_{pp} \end{array}\right], L+U = \left[\begin{array}{c|c|c|c} 0 & \alpha_{12} & \cdots & \alpha_{1p} \\ \hline \alpha_{21} & 0 & \cdots & \alpha_{2p} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline \alpha_{p1} & \alpha_{p2} & \cdots & 0 \end{array}\right]$$

The Jacobi method is equivalent to the Block-Jacobi method, when the block size is equal to 1.

Running the Block-Jacobi method takes longer per iteration than the Jacobi method, as calculating D^{-1} becomes more expensive as the block size increases, however, it can increase the rate of convergence based on the number of non-zero entries that are included within the block-diagonal. Because of this, the Block-Jacobi method is most effective when the non-zero entries of A are mainly concentrated close to the diagonal. Finding an optimal block size often depends on the characteristics of the matrix A, though I will show in Results that there may be other factors that affect this decision as well.

The Jacobi and Block-Jacobi methods will converge when the spectral radius of the iteration matrix is less than one:

$$\rho(D^{-1}(L+U)) < 1$$

Since the iteration matrix varies depending on the block size, this does mean the Block-Jacobi method may converge for certain block sizes, while the Jacobi method does not. One sufficient (though not necessary) condition for convergence for both methods is that A is strictly diagonally dominant, meaning for each row k of A:

$$|a_{kk}| > \sum_{i}^{n} |a_{ki}|, i \neq k$$

5. Implementation

Note: For brevity, some of the following code snippets have been shortened (for example, `assert` statements, and other lines deemed extraneous for the purposes of this paper, have been removed). For the complete source code please refer to the repository [11].

The implementation of this project required multiple steps. First a module for both dense and sparse matrices was defined and implemented in OCaml. Next, basic vector and matrix operations were implemented in sequence and in parallel. Some matrix operations, such as matrix-vector

multiplication, were implemented for both sparse and dense matrices, while others, such as LU-decomposition were implemented only for dense matrices. The next step was to implement the Jacobi method in sequence and in parallel using the matrix modules and linear algebraic operations previously implemented and the final step was to implement the Block-Jacobi method in sequence and in parallel.

5.1. Matrix Modules

The first step is to define and implement types and modules for dense and sparse matrices. A type is a definition of a data structure while a module is a collection of types and functions in OCaml. For this project I implemented seperate `Dense` and `Sparse` modules for dense and sparse matrices respectively. Each includes the type definition for matrices in that format as well as a collection of functions for constructing and using matrices of that type. I included a module for dense matrices both for testing, but also because they are useful when performing LU-decomposition, which is used in the implementation of the Block-Jacobi method.

```
type t = {
  num_rows : int;
  num_cols : int;
  vals : floatarray array;
}
```

Figure 1: Dense Matrix Signature

```
type t = {
  num_rows : int;
  num_cols : int;
  count : int;
  vals : floatarray;
  cols : int array;
  row_ptr : int array;
}
```

Figure 2: Sparse Matrix Signature

Dense matrices are stored using a 2-dimensional array of floats or `floatarray array`, where each `floatarray` represents a row of entries in a matrix. The type signature also includes `int` fields for the number of rows and columns in the matrix (see Figure 1). The module allows a dense matrix to be created from either a `floatarray array` or `float list list`. In either case, each `floatarray` or `float list` must be of the same length. The module includes a `get_val` function of type `t -> int -> int -> float`, a `get_row` function of type `t -> int -> floatarray`, and a `set_val` function of type `t -> int -> int -> float -> unit`, that allow a user to

access and update values in the matrix.

Sparse matrices are stored in Compressed Sparse Row (CSR) format [1] (See Fig 2). In addition to `int` fields for the number or rows and columns, the signature also has a `count` field for the number of non-zero entries in the matrix. The non-zero entries are stored in the `vals` field, a single `floatarray` of length `count`, ordered by row then by column. Each non-zero entry's column index is stored in the `cols` field, an `int array` that corresponds to `vals`. The `row_ptr` field is a second `int array` of length `num_rows`, where each value provides the index where that corresponding row's entries begin in the `vals` and `cols` arrays. See Fig 3 for a simple example.

Figure 3: Example Sparse Matrix in CSR Format

The module provides a `dense_to_sparse` function for creating sparse matrices from dense matrices, as well as a `builder` type. The function `new_builder` of type `int -> int -> builder` takes two arguments for the number of rows and columns and creates a new sparse matrix builder for a matrix of the given dimensions. The function `new_builder` of type `builder -> entry -> builder` adds an `entry` to a `builder` and returns the resulting builder, where `entry` is a type defined as `int * int * float` consisting of a row, column and value. The final function `build_sparse` of type `builder -> t` takes a builder and constructs a sparse matrix from all of its inserted values. The builder type stores an array of entries that is added to everytime a new entry is added. When it is time to construct the sparse matrix from the builder the entries are sorted into their proper order and then the `vals`, `cols`, and `row_ptr` arrays are constructed. In this way, sparse arrays can be easily constructed by reading in row-column-value entries in any order from an external file or other source.

The sparse matrix module also has a `get_val` function similar to the dense matrix module, though it does not contain a `set_val` function as this would require inserting values into the sorted `vals` and `cols` arrays as well as updating index pointers in the `row_ptr` array, which would take

time linear in the number of non-zero entries of the matrix. This functionality is also unnecessary in order to perform the (Block-)Jacobi method, as the matrices don't need to be modified after creation.

For this project, no specific type was defined for storing vectors, as I opted to use the built-in OCaml `floatarray` type instead, which provides all of the necessary functionality. Some additional vector operations were defined which will be detailed in the following section.

5.2. Linear Algebra Operations

The next step is to implement basic vector and matrix operations for our matrix modules. This includes taking vector norms, dot products of two vectors, matrix-vector multiplication, and LU decomposition for block-diagonal matrices. All of these operations are necessary for the Jacobi and Block-Jacobi methods and can be implemented in sequence and in parallel.

5.2.1. Vector Operations Element-wise operations between two vectors can be performed in sequence using the `Float.Array.map2` function [9]. The function has the following signature: `(float -> float -> float) -> floatarray -> floatarray -> floatarray`. According to the documentation: "`map2 f a b` applies function `f` to all the elements of `a` and `b`, and builds a floatarray with the results returned by `f`." For example, adding two vectors can be performed like so, where `(+.)` is the float addition function: `Float.Array.map2 (+.) vec1 vec2`.

We can program a parallel version of `Float.Array.map2` in order to run these same operations in parallel as well. First we initialize the result array in the main thread. Then we split `a` and `b`, (or as I've called them, `vec1` and `vec2`), into `p` subarrays, where `p` is the number of processors. We assign each Domain corresponding subarrays from `vec1` and `vec2`, and apply `f` to all elements of those subarrays. Finally, we copy the resulting subarrays into the results array, and return it once each Domain has finished running.

```
let par_map2 (f : float -> float -> float) (vec1 : floatarray)
      (vec2 : floatarray) (p : int) : floatarray =
  let n = Float.Array.length vec1 in
  let init_domain (i : int) =
    Domain.spawn (fun _ ->
```

```
      let start = i * n / p in
      let len = n / p
      let sub = Float.Array.init len (fun i ->
        let index = i + start in
        let x1 = Float.Array.get vec1 index in
        let x2 = Float.Array.get vec2 index in
        f x1 x2
      ) in
      Float.Array.blit sub 0 results start len
    )
  in
  let domains = List.init p init_domain in
  let _ = List.iter Domain.join domains in
  results
```

It is also possible to utilize higher-order array functions in order to easily implement the dot product:

```
let dot_product (vec1 : floatarray) (vec2 : floatarray) : float =
  Float.Array.fold_left (+.) 0. (Float.Array.map2 ( *.) vec1 vec2)
```

While succinct and correct, this function creates an intermediate array, costing additional space and time, so I opt instead to calculate the dot product using a recursive function:

```
let partial_dot_product (vec1 : floatarray) (start1 : int)
      (vec2 : floatarray) (start2 : int) (n : int) : float =
  let rec aux (i1 : int) (i2 : int) : float =
    if i1 = start1 + n then 0. else
      let x1 = Float.Array.get vec1 i1 in
      let x2 = Float.Array.get vec2 i2 in
      x1 *. x2 +. aux (i1 + 1) (i2 + 1)
  in
  aux start1 start2

let dot_product (vec1 : floatarray) (vec2 : floatarray) : float =
  let n = Float.Array.length vec1 in
  partial_dot_product vec1 0 vec2 0 n
```

I include a `partial_dot_product` function that calculates a dot product of two subvectors starting at indices `start1` and `start2` respectively. This is useful for parallelizing the dot product function, as we can split `vec1` and `vec2` each into `p` subvectors, assign each Domain to calculate the dot product of one pair of subvectors, and then sum the results:

```
let par_dot_product (vec1 : floatarray) (vec2 : floatarray)
      (p : int) : float =
  let n = Float.Array.length vec1 in
  let init_domain (i : int) =
    Domain.spawn (fun _ ->
      let block_size = n / p in
      let start = i * block_size in
      partial_dot_product vec1 start vec2 start block_size
    )
  in
  let domains = List.init p init_domain in
  let results = List.map (Domain.join) domains in
  List.fold_left ( +. ) 0. results
```

With a working dot product function, vector norms can be calculated easily by taking the dot product of a vector with itself as $||x||^2 = x^T x$.

5.2.2. Matrix-Vector Multiplication The dot product function can also be used to easily implement matrix-vector multiplication for dense matrices:

```
let mult_vec (mat_A : Dense.t) (x : floatarray) : floatarray =
  let n = mat_A.num_rows in
  Float.Array.init n (fun i -> dot_product (Dense.get_row mat_A i) x)
```

Implementing matrix-vector multiplication for sparse matrices is slightly more involved, due to the storage format. First I implement a function for multiplying a single row of a matrix by a vector `x`, returning a float:

```
let mult_row_vec (mat_A : Sparse.t) (x : floatarray) (row : int) : float =
  let start_index = Array.get mat_A.row_ptr row in
  let end_index = Array.get mat_A.row_ptr (row + 1) in
  let rec aux (i : int) : float =
    if i = end_index then 0. else
      let m_val = Float.Array.get mat_A.vals i in
      let col = Array.get mat_A.cols i in
      let x_val = Float.Array.get x col in (
      m_val *. x_val +. aux (i + 1)
      )
  in aux start_index
```

This allows the complete matrix-vector multiplication to be calculated easily in both sequence and parallel. For the sequential version I utilize the `Float.Array.init` function [9]:

```
let mult_vec (mat_A : t) (x : floatarray) : floatarray =
  Float.Array.init mat_A.num_rows (mult_row_vec mat_A x)
```

For the parallel version I split `m` into `p` blocks, where `p` is the number of processors and each block consists of `m.num_rows / p` rows of `m`. Then each Domain is assigned to calculate the matrix-vector product of one block of `m` and `b`, similar to how each Domain is assigned a subtask to compute the dot product in parallel, and the results are concatenated.

In addition to implementing `mult_vec` and `par_mult_vec`, I also defined and implemented `mult_LU` and `par_mult_LU` functions which are used for the Jacobi and Block-Jacobi methods. If `mult_vec` and it's parallel counterpart are equivalent to calculating the matrix-vector product Ax, for a matrix A and a vector x, `mult_LU` and it's parallel counterpart are equivalent to splitting A into it's (block-) diagonal, lower-triangular and upper-triangular counterparts for a given block size, such that $A = D + L + U$, and then calculating $(L+U)x$. This is implemented very similarly to regular matrix-vector multiplication, except, when iterating through the non-zero entries of `m`, any value that falls along D is treated as a 0 and skipped.

5.2.3. Extracting D and LU-decomposition The final operations are to extract a dense block-diagonal matrix from a sparse matrix, equivalent to extracting D from A, where $A = D + L + U$, and to perform LU-decomposition on a block-diagonal matrix. The function `diag_block` has type `Sparse.t -> int -> Dense.t array` and takes a sparse matrix `m` and a `block_size`, and returns an array of dense matrices that collectively represent D. The function returns a dense matrix as opposed to a sparse matrix as this makes performing LU-decomposition significantly easier, and the additional cost of storing additional zero entries isn't too expensive as long as the block size remains small in comparison to the overall size of the matrix. The resulting matrix is represented as an array of matrices, where each entry represents a block along the diagonal. This way, instead of storing a single $n \times n$ matrix, we store p $\beta \times \beta$ matrices, where β is the block size and $p = \frac{n}{\beta}$. This function first creates the array of matrices for the block-diagonal and initializes every value to 0. Then it iterates through the non-zero entries of `m`, and if the entry falls along D then the corresponding entry in the resulting array of matrices is updated to match it's value. A similar

function `diag` of type `Sparse.t -> floatarray` is also implemented, where the block size is 1 and the return type is just `floatarray` instead of `Dense.t array` (equivalent to extracting the diagonal of a matrix A) . This function is used for the Jacobi method while `diag_block` is used for the Block-Jacobi method.

To perform LU-decomposition I use the following recursive algorithm [5]. A given square matrix A is divided such that

$$A = \left[\begin{array}{c|c} a & w^T \\ \hline v & A' \end{array}\right],$$

where a is a single entry and v and w are vectors. Now define $c := \frac{1}{a}$, such that

$$A = \left[\begin{array}{c|c} 1 & 0 \\ \hline cv & I_{n-1} \end{array}\right] \left[\begin{array}{c|c} a & w^T \\ \hline 0 & A' - cvw^T \end{array}\right].$$

Now, recursively solve for $A' - cvw^T = L'U'$, such that

$$A = \left[\begin{array}{c|c} 1 & 0 \\ \hline cv & L' \end{array}\right] \left[\begin{array}{c|c} a & w^T \\ \hline 0 & U' \end{array}\right].$$

The function `decomp_LU` has type `Dense.t -> (Dense.t * Dense.t)` and performs LU decomposition using the algorithm described. Although the function runs in sequence, the LU decomposition of a block-diagonal matrix, represented as an array of dense matrices, can be parallelized by splitting the array into p subarrays, then assigning each Domain to map `decomp_LU` across a single subarray, and then concatenating the results, yielding a product of type `(Dense.t * Dense.t) array`.

In addition to this I also implemented functions for solving an LU-decomposition, using substitution, `solve_L`, `solve_U`, and `solve_LU`. `solve_L` initializes the result vector, then recursively solves for each entry, starting from the top of the vector. It solves each entry algebraicly by substituting the already known values, and then updates each entry as it goes.

```
let solve_L (mat_L : t) (b : floatarray) : floatarray =
  let n = mat_L.num_rows in
  let res = Float.Array.make n 0. in
  let rec aux (i : int) : unit =
    if i = n then () else
    let row = get_row mat_L i in
    let a = Float.Array.get row i in
    let b_i = Float.Array.get b i in
    let x = (b_i -. partial_dot_product row 0 res 0 i) /. a in
    Float.Array.set res i x;
    aux (i + 1)
  in
  aux 0;
  res
```

The `solve_U` function is very similar, but starts from the bottom of the solution vector instead, as U is an upper-triangular matrix.

Lastly we implement `solve_LU`, which takes arguments `mat_L`, `mat_U`, and `b`, and solves for $LUx = b$, using `solve_L` and `solve_U`. This is equivalent to solving $Ly = b$ and then $Ux = y$.

```
let solve_LU (mat_L : t) (mat_U : t) (b : floatarray) : floatarray =
  let y = solve_L mat_L b in
  solve_U mat_U y
```

5.3. Implementing the Jacobi Method

The sequential version of the Jacobi method is implemented using two functions, `jacobi_sparse` and a recursive auxiliary function `jacobi_sparse_aux`. The primary function initializes x_0, called `init`, to an array of all zeroes, and calculates D^{-1}, called `d_inv`, by mapping an inverse function across the diagonal of the matrix `m`. The function then calls the auxiliary function, which recursively updates x until it converges, returning the final estimate.

```
let jacobi_sparse (matrix : Sparse.t) (b : floatarray)
      (eta : float) : floatarray =
  let n = matrix.num_rows in
  let init = Float.Array.make n 0. in
  let d_inv = Float.Array.map (fun x -> 1. /. x) (Sparse.diag matrix) in
  jacobi_sparse_aux init
```

The auxiliary function first computes $(L+U)x_k$ using the `Sparse.mult_LU` function implemented earlier, with a block size of 1 (meaning it only skips values on the diagonal). Then it uses that result to calculate $b-(L+U)x_k$, by mapping `(-.)`, or the float subtraction function, across `b` and the result of the previous calculation. Finally it computes $x_{k+1} = D^{-1}(b-(L+U)x_k)$, by multiplying `d_inv` by the result of the previous calculation. Since `d_inv` is just the inverse of the diagonal, represented by a `floatarray`, it does this by mapping `(*.)`, or the float multiplication function, across `d_inv` and the result of the previous calculation. Finally, the function checks convergence by calculating the $||x_{k+1}-x_k||^2$. The difference of the two vectors can be calculated using a map function, as already described, and the squared norm is calculated using the dot product. If the squared norm is below some `eta`, specified by the user, then the resulting vector is returned. Otherwise, the function is called again using the new vector for x_{k+1}.

```
let rec jacobi_sparse_aux (x : floatarray) : floatarray =
  let lu_x = Sparse.mult_LU matrix x 1 in
  let b_minus = Float.Array.map2 ( -. ) b lu_x in
  let new_x = Float.Array.map2 ( *. ) d_inv b_minus in
  let diff = Float.Array.map2 ( -. ) new_x x in
  let diff_sq = dot_product diff diff in
  if diff_sq < eta then new_x else jacobi_sparse_aux new_x
```

Here I've shown slightly simplified versions of the `jacobi_sparse` and `jacobi_sparse_aux` functions. The actual `jacobi_sparse` function also calculates and reports the squared error of the final estimate ($||Ax-b||^2$). The `jacobi_sparse_aux` also has a parameter for the maximum number of iterations it is to perform, and also checks that $||x_{k+1}-x_k||^2$ doesn't get larger from one iteration to the next, as this implies that the solution does not converge. If the solution diverges, the function raises an exception.

The parallel version is essentially the same, except it uses the parallel version of the `mult_LU`, `map2`, and `dot_product` functions that were implemented earlier, as opposed to the sequential versions.

5.4. Implementing the Block Jacobi Method

The Block Jacobi method also uses two functions, `block_jacobi` and `block_jacobi_aux`, similar to the Jacobi method. `block_jacobi` is very similar to `jacobi_sparse` with the only difference that it takes an extra argument for the block size, extracts the block diagonal of `matrix` as opposed to just the diagonal, and calculates the LU-decomposition of the block diagonal by mapping our LU-decomposition function across an array of matrices, as described earlier.

```
let block_jacobi (matrix : Sparse.t) (b : floatarray)
      (block_size : int) (eta : float) : floatarray =
  let n = matrix.num_rows in
  let init = Float.Array.make n 0. in
  let d_block = Sparse.diag_block matrix block_size in
  let d_LU = Array.map Dense.decomp_LU d_block in
  block_jacobi_aux init
```

We choose to take the LU-decomposition of the block diagonal, as it saves us from having to calculate the inverse directly. Instead of taking the inverse and calculating $x_{k+1} = D^{-1}(b-(L+U)x_k)$, we instead decompose D into the product of D_L and D_U. Then we can solve the equation $D_L D_U x_{k+1} = b-(L+U)x_k$, using the `Dense.solve_LU` function. Since D_L and D_U are both triangular matrices, this can be done each iteration with relative efficiency.

As the LU-decomposition of D is stored as a type `(Dense.t * Dense.t) array`, we define a function `solve_block`, that accesses a specific block of the decomposition of D and solves it for the corresponding subvector of `b`.

```
let solve_block (b : floatarray) (i : int)
      (mat_LU : Dense.t * Dense.t) : floatarray =
  let b_sub = Float.Array.sub b (i * block_size) block_size in
  let mat_L, mat_U = mat_LU in
  Dense.solve_LU mat_L mat_U b_sub
```

We use this function in `block_jacobi_aux`, to solve for the LU-decomposition of D, by mapping it across `d_LU`. This returns an array of subvectors which we concatenate to form `new_x`. Otherwise, `block_jacobi_aux` looks very similar to `jacobi_sparse_aux`.

```
let rec block_jacobi_aux (x : floatarray) : floatarray =
  let lu_x = Sparse.mult_LU matrix x block_size in
  let b_minus = Float.Array.map2 ( -. ) b lu_x in
  let x_subs = Array.mapi (solve_block b_minus) d_LU in
  let new_x = Float.Array.concat (Array.to_list x_subs) in
  let diff = Float.Array.map2 ( -. ) new_x x in
  let diff_sq = dot_product diff diff in
  if diff_sq < eta then new_x else block_jacobi_aux new_x
```

Once again, we can parallelize this very easily by using our parallel versions of the vector and matrix operations implemented earlier as opposed to the sequential versions.

6. Results and Evaluation

In order to measure the effectiveness of my parallel implementations, I ran basic timing tests for vector operations, matrix-vector multiplication, and the Block-Jacobi method with various block sizes. The tests were all ran on the Princeton Computer Science Department's Cycle Server, which runs on Linux, with a x86_64 architecure, and allows for shared-memory parallelization across 80 processors, with two sockets, 20 cores per socket and two threads per core. Each test was performed with 1, 2, 4, 8, 16, 32, and 64 domains for comparison.

6.1. Vector Operations

I tested the parallel implementation of the dot product and the `map2` functions on vectors of length $100,000$, $1,000,000$, and $10,000,000$. For the dot product I calculated the dot product of the vector with itself, and for `map2` I calculated the vector sum of the vector with itself. Each calculation was performed one hundred times, and the total duration was measured in seconds. The results are displayed in the tables below and a graph can be seen in Fig 4 (note the log-scale on both axes).

# Processors:	1	2	4	8	16	32	64
100K	0.656	0.797	0.301	0.581	0.799	0.959	1.320
1M	8.50	5.30	3.96	2.69	2.66	3.93	4.97
10M	398.0	125.1	43.8	18.8	10.5	11.6	25.7

Table 1: Dot Product Results

# Processors:	1	2	4	8	16	32	64
100K	0.573	0.598	0.466	0.499	0.613	0.911	2.574
1M	3.16	2.59	2.22	2.02	2.42	4.82	4.12
10M	19.0	14.9	9.91	8.32	8.07	8.95	8.10

Table 2: Vector Addition Results

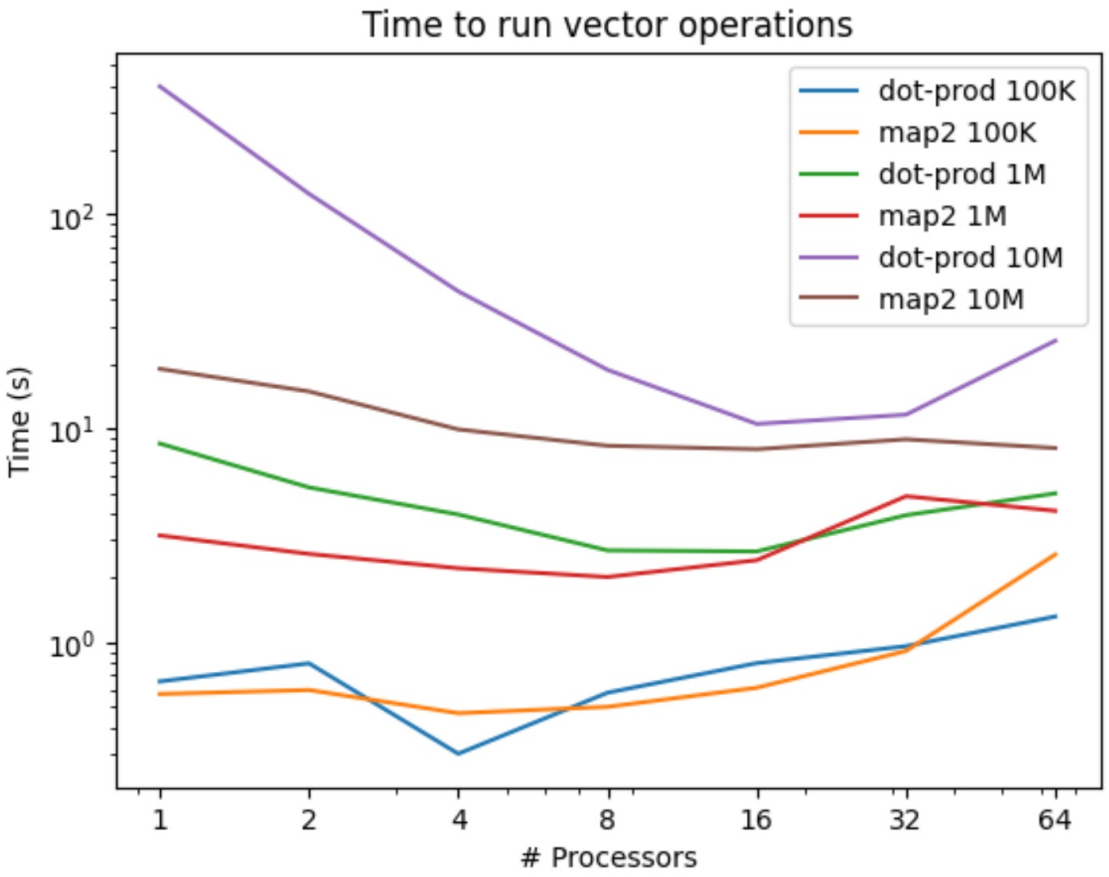

Figure 4

One might notice right away that doubling the number of processors generally doesn't halve the time necessary to complete the calculation (the exception is calculating the dot product for vectors of size $n = 10,000,000$, in which case we initially see speed-up by more than a factor of 2). In fact, for smaller vectors, increasing the number of processors actually has a very negligible effect. However, as the size of the vector increases, the speed-up becomes more pronounced. This suggests that instantiating Domains in OCaml has some amount of overhead, that becomes more apparent when the size of the calculation assigned to that Domain is small. We also notice that the dot product seems to get better speed-up compared to the `map2` function. This may be due to

the fact that the `map2` function requires instantiating a new array (which can't be parallelized) and then writing to that array in parallel, where as the dot product is strictly computational and doesn't require writing to memory.

6.2. Matrix-Vector Multiplication

For the following two sections, I used three matrices from the SuiteSparse Matrix Collection [7], `cage12`, `cage13`, and `cage14`. All three are asymmetric matrices that come from directed weighted graphs. `cage12` has dimensions $130,228 \times 130,228$ and $2,032,536$ non-zero entries. `cage13` has dimensions $445,315 \times 445,315$ and $7,479,343$ non-zero entries. `cage14` has dimensions $1,505,785 \times 1,505,785$ and $27,130,349$ non-zero entries.

To test matrix-multiplication, I generated vectors of the appropriate length and timed how long it takes to compute 100 matrix-vector multiplications. The data can be seen in the table below and graphed in Fig 5. Once again, we notice a speed-up by less than a factor of 2 whenever the number of processors doubles, although the speed-up is greater when the overall size of the calculation increases.

# Processors:	1	2	4	8	16	32	64
cage12	2.55	1.98	1.82	1.51	1.28	1.22	1.84
cage13	7.43	5.02	4.07	3.35	2.92	3.07	2.94
cage14	23.3	11.9	9.95	7.65	6.19	6.85	7.63

Table 3: Matrix-Vector Multiplication Results

6.3. Block-Jacobi Method

In order to test the Jacobi and Block-Jacobi methods, I ran the parallel implementation of the Block-Jacobi method with the ones-vector of the appropriate length for b, $\varepsilon = 10^{-6}$ and block sizes of 1 (equivalent to the Jacobi method), 4, 16, 64, and 256, and measured the number of iterations it took to converge and the duration in seconds. The results are shown in the tables below.

Here we see some very informative results. As expected, we see that increasing the block size decreases the number of iterations required to converge. Consistent with our previous results, we

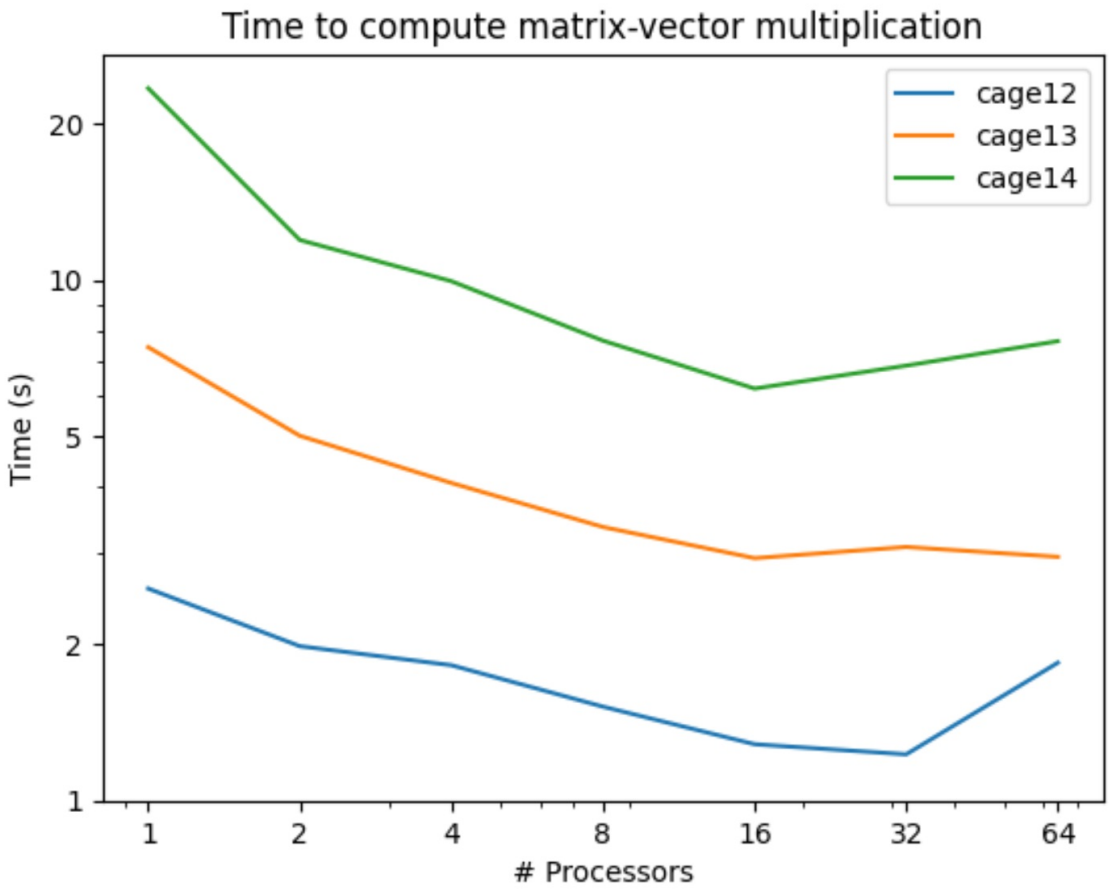

Figure 5

Block Size:	1	4	16	64	256
cage12	176	101	49	26	17
cage13	196	126	65	33	21
cage14	181	103	67	36	22

Table 4: Number of Iterations for Convergence

# Processors:	1	2	4	8	16	32	64
Block Size 1	6.814	6.886	6.283	5.828	6.138	5.622	6.770
Block Size 4	6.22	5.736	5.125	4.646	4.369	4.669	5.721
Block Size 16	3.607	3.145	3.091	2.517	2.470	2.509	3.067
Block Size 64	4.393	3.196	2.536	2.350	1.917	2.439	2.708
Block Size 256	11.50	7.195	4.467	3.306	2.695	3.357	4.194

Table 5: cage12 Timing Results

also see that `cage14`, the largest matrix, has the most noticeable speed-up from increasing the number of processors, while `cage12` has the smallest speed-up. Interestingly, we can see that the optimal block size depends in part on the number of processors used. For all three matrices, 16

# Processors:	1	2	4	8	16	32	64
Block Size 1	26.77	20.54	16.04	14.68	14.93	16.94	16.37
Block Size 4	24.10	18.32	15.28	14.07	14.28	15.25	15.43
Block Size 16	15.09	10.75	8.606	7.222	7.752	7.892	8.049
Block Size 64	18.51	11.18	7.352	5.470	5.120	5.220	5.930
Block Size 256	35.21	26.66	15.95	10.21	8.685	7.874	7.166

Table 6: cage13 Timing Results

# Processors:	1	2	4	8	16	32	64
Block Size 1	77.58	54.71	38.84	29.43	27.53	29.16	28.98
Block Size 4	63.45	43.36	31.00	24.35	23.68	26.14	26.25
Block Size 16	51.16	32.55	21.89	16.59	15.18	16.01	17.02
Block Size 64	57.83	38.21	22.72	14.94	12.10	12.94	12.23
Block Size 256	119.2	76.54	51.42	34.48	24.97	25.00	22.17

Table 7: cage14 Timing Results

seems to be the best block size of those tested when only using a single processor, however, a block size of 64 seems to do better when the number of processors increases. In addition to this, we notice that the speed-up from increasing the number of processors is greater with larger block sizes. These results seem to suggest that the Block-Jacobi method may parallelize better than the Jacobi method, and that the optimal block-size when running the Block-Jacobi method may increase as the number of processors increases.

7. Future Work

There are multiple opportunities for future work with this project. One opportunity could be exploring the effect of parallelization on the Block-Jacobi method further. My results seem to indicate that the Block-Jacobi method parallelizes better (meaning it gets better speed-up as the number of processors increases) as the block size increases. As a result of this, the optimal block size depends not just on the matrix but also on the number of processors used for the calculation. This is a very interesting and useful result if true. Further testing could include getting timing results for more block sizes and other matrices.

Another opportunity for further work could be using the module for sparse matrices that I've built with the accompanying linear algebra operations to implement other algorithms. For example,

there are other iterative methods for solving linear systems, such as the Gauss-Seidel method or Conjugate-Gradient method. Both of these algorithms have their own benefits for using, but would come with their own sets of challenges. For example, the Gauss-Seidel has guaranteed convergence when A is symmetric positive-definite, a property that the Jacobi and Block-Jacobi methods do not have. However, it is much more difficult to parallelize, in comparison to the Jacobi methods, because it updates each entry of x_{k+1} one at a time, instead of all at once.

As mentioned earlier, there are a number of open source libraries that currently exist for scientific computing and numerical linear algebra in OCaml. Another opportunity for furthering the research started in this project could be using the module that I built to add support for sparse matrices to one of these libraries such as Owl [14], which already offers support and basic operations for dense matrices. One more opportunity could be utilizing existing libraries for parallel computing in OCaml such as DomainsLib [12], kCas [13], or MoonPool [6], to see if it's possible to improve parallel performance.

8. Conclusion

With this project I hope to have met all of the goals outlined at the start of this paper. These were to build a module for using and performing calculations in parallel with sparse matrices in OCaml, to implement the Jacobi and Block-Jacobi methods using said module, and to measure the effectiveness of parallelization using Domains in OCaml v5 for these applications.

The implemented sparse matrix module allows for easy and efficient construction and storage of sparse matrices using CSR format. It provides support for basic linear algebra operations including matrix-vector multiplication and dot products, which can be run in parallel on multicore hardware using the Domains library. It also provides support for a parallel version of the `map2` function which can be used to perform operations such as vector addition and element-wise vector multiplication. This module was used to implement the Jacobi and Block-Jacobi methods, which can efficiently calculate approximate solutions to arbitrary precision for some linear systems of the form $Ax = b$, where A is a square matrix with real values, including any system where A is diagonally-dominant.

Basic timing tests show that it is possible to achieve significant speed-up by using the Domains library and running with multiple processors in OCaml v5. However, the degree of speed-up depends on the individual task. Specifically, large tasks that involve fewer writes to memory tend to see the fastest speed-up, and even in the best cases, parallelization rarely leads to a one-to-one increase in speed, where doubling the number of processors halves the computation time. Additionally, timing results from the Block-Jacobi method indicate that the method parallelizes better as the block size increases, and as a result that the optimal block-size for a given problem may also increase as the number of processors increases. This result could have meaningful implications in the field of numerical linear algebra and deserves further investigation.

9. Honor Statement

This paper represents my own work in accordance with University policy.

- Samuel Sanft

References

[1] R. Barret, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM Publications Library, 1994.

[2] G. E. Blelloch, "Programming parallel algorithms," *Communications of the ACM*, pp. 85–97, 1996.

[3] G. E. Blelloch and B. M. Maggs, *Parallel Algorithms*. Chapman & Hall/CRC, 2010, ch. 25.

[4] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, pp. 107–117, 1998.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[6] S. Cruanes, "moonpool," 2023. [Online]. Available: https://github.com/c-cube/moonpool

[7] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. on Mathematical Software*, pp. 1–25, 2011.

[8] OCaml, "Ocaml library : Domain." [Online]. Available: https://v2.ocaml.org/api/Domain.html

[9] ——, "Ocaml library : Float.array." [Online]. Available: https://v2.ocaml.org/api/Float.Array.html

[10] ——, "Ocaml 5.0.0," December 2022. [Online]. Available: https://ocaml.org/releases/5.0.0

[11] S. Sanft, "Iw spring 2024 source code." [Online]. Available: https://github.com/ss7886/IW-Spring24

[12] K. Sivaramakrishnan, "Domainslib," 2023. [Online]. Available: https://github.com/ocaml-multicore/domainslib

[13] ——, "kcas," 2024. [Online]. Available: https://github.com/ocaml-multicore/kcas

[14] L. Wang, "Owl," 2023. [Online]. Available: https://github.com/owlbarn/owl

[15] L. Wang, J. Zhao, and R. Mortier, *OCaml Scientific Computing: Functional Programming in Data Science and Artficial Intelligence*. Springer, 2022.