UNIVERSITY OF CALIFORNIA
Santa Barbara

# An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Viral B. Shah

June 2007

The Dissertation of
Viral B. Shah is approved:

---

Professor Alan Edelman

---

Professor Linda Petzold

---

Professor Frédéric Gibou

---

Professor John R. Gilbert, Committee Chairperson

June 2007

An Interactive System for

Combinatorial Scientific Computing

with an Emphasis on Programmer Productivity

To my parents, and Darshan.

# Acknowledgements

I was extremely lucky to carry out my dissertation work at the beautiful UCSB campus. I am glad to have been part of the computational sciences and engineering program at UCSB, which provided me with the required training to engage in cross-disciplinary research.

First, I would like to thank my committee for commenting on my manuscript, which significantly improved the quality of presentation in this thesis. This thesis would not have been possible without the superb guidance and mentorship of my advisor, John Gilbert. I will always remain deeply indebted to him for teaching me the art and science of conducting research. This thesis owes a great deal to his limitless patience. I am also grateful to him for finding hundreds of typographical errors in my manuscript. The course for this thesis was set during my visit to MIT, hosted by Alan Edelman. His constant feedback and suggestions greatly improved the work presented in this thesis. They both deeply influenced the philosophical underpinnings of this work.

Research is a collaborative effort; I thoroughly enjoyed working with my co-authors. My collaboration with Brad McRae on modeling landscape connectivity was extremely fruitful, and a great learning experience. The chapter on landscape connectivity would be nowhere close to its current form without Brad's extensive feedback. I would like to thank Vikram Aggarwal for helping with the analysis of productivity data, the implementation of support graph preconditioners, and for providing me with quadtree meshes. I would like to thank Min Roh for our work on solving linear systems arising from quadtree meshes. The work on parallel sorting was done with David Cheng at MIT. David did a great job implementing the algorithm in its current form. The work in computational fluid dynamics to model porous media flows was done in collaboration with Nisheet Goyal. The timed Markov modeling of programmer productivity was done in collaboration with Burton Smith and David Mizell of Cray.

I had several fruitful discussions on parallel Matlab with Parry Husbands, Ron Choy, Per-Olof Persson, Sudarshan Raghunathan, Jeremy Kepner, and Nadya Travinin-Bliss. I also benefited a lot from discussions with Lorin Hochstein, Jeremy Kepner, and Imran patel on programmer productivity. I am grateful to Esmond Ng and Parry Husbands for having me at Lawrence Berkeley National Labs as a summer intern in 2004, and to the folks at Interactive Supercomputing for hosting me in the summers of 2005 and 2006.

I would like to thank the Department of Computer Science at UCSB, and the DARPA/DOE HPCS program for funding my dissertation work. Several institutions provided me with computing facilities: Interactive Supercomputing, SGI, the National Center for Ecological Analysis and Synthesis at UCSB, CISE/IGERT at UCSB, and the San Diego Supercomputer Center. I am thankful to the students in several parallel computing classes at UCSB who gave consent for their activities to be monitored, and the UCSB Office of Research for providing the required human test subject clearance to conduct the classroom experiments.

I would like to thank Ajay Shah, Susan Thomas, and Ashok Srinivasan for getting me interested in scientific computing during my undergraduate studies. They were great mentors to me, and it is because of their early guidance that I decided to pursue graduate studies in parallel scientific computing.

Lastly, this thesis would have not happened without the loving support of my family. The ultimate frisbee community at UCSB and in Santa Barbara made sure I got enough exercise, and the beaches around UCSB provided great surf. It would have been hard to focus on research without these distractions. I would especially like to thank my girlfriend Swapnika for religiously driving over weekends from Los Angeles to see me. I cannot imagine having achieved this without her support and company.

This Ph.D. has been a great journey, and I would never trade this experience for anything else.

# Curriculum Vitæ

Viral B. Shah

**Education**

|  |  |
|---|---|
| 2001 | Bachelor of Engineering, University of Mumbai. |

**Experience**

|  |  |
|---|---|
| 2002 – 2007 | Graduate Research Assistant, UC Santa Barbara. |
| 2005 – 2007 | Consultant, Interactive Supercomputing LLC. |
| 2004 | Intern, Lawrence Berkeley National Laboratory. |

**Selected Publications**

V. B. Shah, J. R. Gilbert. Sparse matrices in Matlab*P: Design and implementation. *Proceedings of High Performance Computing*, 2004.

J. R. Gilbert, S. Reinhardt, V. B. Shah. High performance graph algorithms from parallel sparse matrices. *Proceedings of Workshop on State of the art in Scientific and Parallel Computing*, 2006.

A. Funk, J. R. Gilbert, D. Mizell, V. Shah. Modeling programmer workflows with timed Markov Models. *Cyberinfrastructure Technology Watch*, Nov 2006.

D. Bader, K. Madduri, J. R. Gilbert, V. Shah, J. Kepner, T. Meuse, A. Krishnamurthy. *Cyberinfrastructure Technology Watch*, Nov 2006.

# Abstract

## An Interactive System for Combinatorial Scientific Computing with an Emphasis on Programmer Productivity

Viral B. Shah

Two trends have emerged of late in scientific computing. The first one is the adoption of high level interactive programming environments such as Matlab, R and Python. This is largely due to diverse communities in physical sciences, engineering and social sciences using the computational experiment to supplement results from theory and experiments. Graphs are increasingly being used to model relationships between individual elements in many physical systems. Computation on graphs combined with numerical simulation represents the other trend in scientific computing.

This thesis describes the design and implementation of a unified platform for combinatorial and numerical computing in Star-P, a parallel implementation of the Matlab programming language. Sparse matrices allow structured representation of irregular data structures, decompositions, and irregular access patterns in parallel applications. The duality between sparse matrix algorithms and graph algorithms is used to build a toolbox to compute with graphs in Star-P.

Circuitscape, a tool in landscape ecology, models animal movement and gene flow as resistive networks. Initially, computation on a moderate sized landscape required three days. The combination of distributed sparse matrices in Star-P, the graph toolbox, and vectorization allow the same computation to be performed within fifteen minutes. Much larger landscapes can now be processed within a few hours.

One of the salient features of interactive environments such as Matlab and Star-P is increased programmer productivity. This thesis describes the design of experiments to collect productivity data from programmers. The concept of replay is presented, which makes these experiments reproducible. The replay also allows researchers to extract data from variables which were not observed when the experiment was in progress. Data collected from two classroom experiments is then used to construct timed Markov models of programmer productivity.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Programming environments such as MATLAB [91], R [74], and Python [121] are becoming popular for numerical computations. Their dynamic nature, interactivity, and high level domain specific abstractions make them very popular among scientists and engineers. Scientific computing is no longer practised only by engineers and physicists, as has been the case until recently. Several diverse research communities now include computation in their research methodology, on par with theory and experiments. In the sciences, users of computational techniques include biologists, chemists, ecologists, mathematicians, and statisticians. Computational methods are also becoming popular in the social sciences, such as economics, finance, sociology, and psychology. As a result, modern applications require a wider variety of computational techniques.

Traditionally, numerical computing has been the focus of scientific computing. MATLAB has made numerical computing accessible to scientists and engineers who are not necessarily experts in numerical computing. Clusters have made it possible for anyone with a modest budget to buy a small supercomputer. The advent of multicore CPUs has brought parallel computing to desktops. However, advances in sequential scientific computing have yet to be realized in the case of high performance computing.

Emerging high performance applications such as web search, information retrieval, data mining, knowledge discovery, bioinformatics, computational biology, multiscale modeling, geometric modeling, etc. have a combinatorial aspect in addition to the numerical aspect of their problem solving. Breakthroughs in such areas require a better way to program high performance computers, combining modern numerical and combinatorial tools in one easy to use package.

A parallel implementation of the matlab programming language such as Star-P is a step in the right direction. Star-P [72] does not require the user to be an expert on parallel computing, numerical computing, or combinatorial computing. It extends the matlab language with only a handful of operators and commands, which makes it easy to use for anyone familiar with Matlab. The result is a system that allows for very high efficiencies in time to solution while simultaneously harnessing the power of modern high performance computers.

This thesis will address both parts of the puzzle:

1. Design an easy to use programming environment for high performance numerical and combinatorial computing. (Part I).

2. Measure the gains in productivity realized through such a system. (Part II).

Modern languages for numerical computing provide sparse arrays as a basic data structure. In MATLAB sparse matrices are first class objects, since almost any operation that can be performed on a dense matrix can be performed on a sparse matrix. Chapter 3 describes the design and implementation of sparse matrix support in STAR-P. Sparse matrix computations allow structured representation of irregular data structures, decompositions, and irregular access patterns in parallel applications. One of the innovative contributions of this thesis is an outline for the construction of an infrastructure for numerical and combinatorial computing on top of a sparse array implementation.

First, a suitable data structure and data decomposition for sparse matrices are fixed. The data is stored in compressed rows, while the sparse matrix itself is split by rows across processors. No other layouts or distributions are supported deliberately to keep the implementation robust and simple. All operations such as matrix constructors, arithmetic, matrix multiplication, and indexing are then designed to provide good performance for these design decisions. One operation that a system such as Star-P needs to perform frequently is a test for matrix symmetry.

Sparse matrix multiplication is another very important primitive. Sparse matrix dense vector multiplication is a well studied problem, and its efficient implementation in Star-P allows access to a variety of Krylov subspace algorithms for solutions of linear systems and eigenvalue problems. Sparse matrix matrix multiplication, on the other hand, is a very useful primitive for combinatorial computing.

Sparse matrices and graphs are duals of each other. This makes it possible for graph algorithms in Star-P to be implemented as operations on sparse matrices. Thus the numerical computing infrastructure in Star-P is leveraged to provide a comprehensive infrastructure for combinatorial computing. Chapter 3 describes some parallel graph algorithms in detail. The combinatorial computing capability is built as a toolbox, which provides several graph querying capabilities, graph operations and visualization tools.

Sorting is another primitive that is commonly used in computations on irregular data. A lot of the sparse matrix support in Star-P is built upon parallel sorting. Chapter 4 describes the parallel sorting algorithm used in Star-P for general purpose sorting. Sorting is used very often in a variety of ways to aid combinatorial computing, mainly to generate orderings which may expose the combinatorial structure of the underlying data.

The usefulness of such an infrastructure for building higher level tools is described in Chapter 5. The graph toolbox is used to implement SSCA#2, a graph analysis benchmark. Chapter 5 also describes the implementation of combinatorial preconditioners and non-negative matrix factorizations. Preconditioners accelerate the solution of linear systems by iterative methods, whereas non-negative matrix factorizations are useful for pattern discovery.

Chapter 6 describes an application in Circuitscape, an application in computational ecology. Dispersal is a key process in maintaining healthy and viable animal populations. Circuitscape models a landscape as a graph, specifically a resistive network. The computation has a combinatorial part which involves manipulating very large graphs, and a numerical part, which involves solving large sparse linear systems to compute effective resistance. The combination of distributed sparse matrices in Star-P, the graph toolbox, and vectorization allow computations on large landscapes, which wasn't possible earlier.

So far, intuition has been the basis of most programming language design. It would be nice to convert this art to science. To this end, methods to collect and analyze data from programmers are proposed, in order to say something meaningful about their productivity.

Data was gathered from two instances of our parallel computing class at UC Santa Barbara in 2004 and 2006. In 2004, the students programmed a parallel sorting algorithm in MPI. In 2006, they programmed the game of Life. Half the class used MPI, while the other half used UPC. We instrumented the compile and runtime environments to collect data such as timestamps, durations, code snapshots etc.

The experimental setup evolved quite a bit from 2004 to 2006. In 2004, the instrumentation had a questionnaire that prompted the student at every compile. It turns out that the compile reason can be guessed automatically using the replay. The replay is a process in which every snapshot of a student code is rerun, and all runtime data, including program state (correctness, failure, crash) is captured. The replay essentially allows the experimenter to go back and re-examine the experiment in a fully controlled environment.

Since the 2004 data gathering proved that questionnaires are redundant, the 2006 experiment instead focused on providing harnesses to students which include data generators, validators, Makefiles, etc. This experiment was designed to allow perfect replay.

Once the data is collected, a model is needed to interpret it. One such model is a Timed Markov model. A programmer's workflow may be thought of as a directed graph. The nodes of the graph represent activities such as run, debug, optimize, compile etc. The edges represent transitions between activities. This thesis provides a study of students programming in UPC and C/MPI, comparing and contrasting their workflows. This is one model which may allow quantitative analysis of different programming languages.

It is important that a language designer worry about programmer productivity. Language design and programmer productivity are not isolated problems; they go hand in hand. Better productivity metrics provide better feedback to the language designer, which in turn leads to a system which allows higher levels of productivity.

# Part I

# Sparse Matrices and Graphs

# Chapter 2

# Sparse Matrices in Star-P: Design and Implementation

## 2.1 Introduction

The numerical computing community has a strong discipline of developing high quality library codes. EISPACK [113] and LINPACK [42] are some of the oldest numerical libraries, developed in the 1960s. They have since been superseded by BLAS [19] and LAPACK [7]. Today, libraries such as GotoBLAS [63], ATLAS [126], FFTW [51], and OSKI [122] provide efficient implementations of commonly used routines in scientific computing. Netlib [26] maintains a comprehensive list of mathematical software and papers.

Matlab [91] is a widely used tool in scientific computing. It started in the 1970s as an interactive interface to EISPACK and LINPACK. Today, Matlab provides a fully featured programming environment encompassing modern numerical libraries such as ATLAS [126], LAPACK [7], FFTW[51], rich graphics capabilities for visualization, and several domain specific toolboxes.

The landscape of computing has changed a lot in the last few years. Higher performance no longer comes from faster clock speeds, but from newer architecture breakthroughs. Clusters became increasingly common in the 1990s, and shared memory architectures are making a comeback. Multi-core CPUs are already in use, with more and more processor cores being packed together in a single CPU socket.

The question naturally arises: How does one program these computers ? MATLAB is widely used by non-computer scientists, and more so by those who are not experts at technical or parallel computing. Such programmers would prefer to use the tools they are familiar with on newer architectures. Several attempts were made to create a parallel MATLAB [32]. We believe STAR-P [76] is the most promising of all such approaches. STAR-P was originally an academic project at MIT and UCSB, and has since been commercialized. Much of the work described in this thesis has been incorporated into STAR-P, or built on the STAR-P platform.

The basic design of the STAR-P system and operations on dense matrices have been discussed in earlier work [33, 72, 73]. We will focus on the design of the sparse matrix infrastructure in STAR-P, with the eventual goal to use it as a common platform for numerical and combinatorial computing. We will specifically not focus on task parallel computing in Star-P, which in the past has been referred to as `MM-mode` and is now called `ppeval`. Briefly, the idea behind `ppeval` is to express iteration without loops, and instead focus on data [125]. The original `MM-mode` was extended to use a global address space, and this was referred to as `GAS-MM`. We will restrict our focus to data parallel computing with distributed sparse matrices.

Sparse matrices may have dimensions that are in millions or hundreds of millions, and enough non-zeros that they cannot fit on a single workstation. Sometimes, the sparse matrices are themselves not too large, but due to the fill caused by intermediate operations (for e.g. LU factorization), it becomes necessary to distribute the factors over several processors. The goal of sparse matrix support in Star-P is to allow users to perform operations on sparse matrices as transparently as in Matlab.

It is true in Matlab, as well as in Star-P, that many key operations are provided by public domain software (linear algebra, solvers, fft, etc.). Apart from simple operations such as array arithmetic, Matlab allows matrix multiplication, array indexing, array assignment and concatenation of arrays, among other things. These operations form extremely powerful primitives upon which other functions, toolboxes, and libraries are built. The challenge in the implementation lies in selecting the right data structures and algorithms which implement all operations efficiently, allowing them to be combined in any number of ways.

## 2.2 Sparse matrices: A user's view

In addition to Matlab's sparse and dense matrices, Star-P provides support for distributed sparse (dsparse) and distributed dense (ddense) matrices.

The *p* operator provides for parallelism in Star-P. For example, a random parallel dense matrix (ddense) distributed by rows across processors is created as follows:

```
>> A = rand (100000*p, 100000)
```

Similarly, a random parallel sparse matrix (dsparse) also distributed across processors by rows is created as follows: (The third argument specifies the density of non-zeros):

```
>> S = sprand (1000000*p, 1000000, 0.001)
```

We use the overloading facilities in Matlab to define a *dsparse* object. The Star-P language requires that almost all (meaningful) operations that can be performed in Matlab be possible with Star-P. Our implementation provides a working basis, but is not quite a drop–in replacement for existing Matlab programs. The rest of this chapter will describe the design decisions taken, trade-offs made, and lessons learnt.

STAR-P achieves parallelism through polymorphism. Operations on ddense matrices produce ddense matrices. But, once initiated, sparsity propagates. Operations on dsparse matrices produce dsparse matrices. An operation on a mixture of dsparse and ddense matrices produces a dsparse matrix unless the operator destroys sparsity. The user can explicitly convert a ddense matrix to a dsparse matrix using `sparse (A)`. Similarly a dsparse matrix can be converted to a ddense matrix using `full (S)`. A dsparse matrix can also be converted into a MATLAB sparse matrix using `ppfront(S)`.

## 2.3 Data Structures and Storage

Compressed row and column data structures have been shown to be efficient for sparse linear algebra [66]. MATLAB stores sparse matrices on a single processor in a Compressed Sparse Column (CSC) data structure [56]. The STAR-P language allows ddense matrices to be distributed by block rows or block columns [33, 72]. Our implementation supports only the block row distribution for dsparse matrices. This is a design choice to prevent the combinatorial explosion of argument types. Block layout by rows makes the Compressed Sparse

Row data structure a logical choice to store the sparse matrix slice on each processor. The choice to use a block row layout is not arbitrary, but the reasoning is as follows:

- The iterative methods community largely uses row based storage. Since we believe that iterative methods will be the methods of choice for large sparse matrices, we want to ensure maximum compatibility with existing code.
- A row based data structure also allows efficient implementation of "matvec" (sparse matrix dense vector product), the workhorse of several iterative methods such as Conjugate Gradient and Generalized Minimal Residual.

For the expert user, storing sparse matrices by rows instead of by columns changes the programming model. For instance, high performance sparse matrix codes in MATLAB are often carefully written so that all accesses into sparse matrices are by columns. When run in STAR-P, such codes may display different performance characteristics, since dsparse matrices are stored by rows.

The CSR data structure stores whole rows contiguously in a single array on each processor. If a processor has $nnz$ non-zeros, CSR uses an array of length $nnz$ to store the non-zeros and another array of length $nnz$ to store column

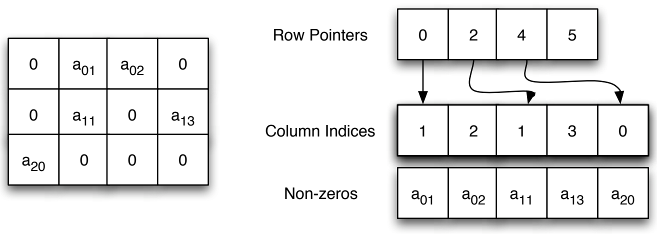


**Figure 2.1:** The matrix is shown in its dense representation on the left, and its compressed sparse rows (CSR) representation on the right. In the CSR data structure, non-zeros are stored in three vectors. Two vectors of length $nnz$ store the non-zero elements and their column indices. A vector of row pointers marks the beginning of each new row in the non-zero and column index vectors.

indices, as shown in Figure 2.1. Row boundaries are specified by an array of length $m + 1$, where $m$ is the number of rows on that processor.

Using double precision floating point values for the non-zeros on 32-bit architectures, an $m \times n$ real sparse matrix with $nnz$ non-zeros uses $12nnz + 4(m+1)$ bytes of memory. On 64-bit architectures, it uses $16nnz + 8(m + 1)$ bytes. STAR-P supports complex sparse matrices as well. In the 32-bit case, the storage required is $20nnz + 4(m + 1)$ bytes, while it is $24nnz + 8(m + 1)$ bytes on 64-bit architectures.

## 2.4 Operations on distributed sparse matrices

The design of sparse matrix algorithms in STAR-P follows the same design principles as in MATLAB [56].

1. Storage required for a sparse matrix should be $O(nnz)$, proportional to the number of non-zero elements.

2. Running time for a sparse matrix algorithm should be $O(flops)$. It should be proportional to the number of floating point operations required to obtain the result.

The data structure described in the previous section satisfies the requirement for storage. The second principle is difficult to achieve exactly in practice. Typically, most implementations achieve running time close to $O(flops)$ for commonly used sparse matrix operations. For example, accessing a single element of a sparse matrix should be a constant time operation. With a CSR data structure, it typically takes time porportional to the logarithm of the length of the row to access a single element. Similarly, insertion of single elements into a CSR data structure generates extensive data movement. Such operations are efficiently performed with the `sparse/find` routines (described next), which work with triples rather than individual elements.

## 2.4.1 Constructors

There are several ways to construct distributed sparse matrices in STAR-P:

1. `ppback` converts a sequential MATLAB matrix to a distributed STAR-P matrix. If the input is a sparse matrix, the result is a dsparse matrix.

2. `sparse` creates a sparse matrix from dense vectors giving a list of non-zero values. A distributed sparse matrix is automatically created, if the dense vectors are distributed. `find` is the dual of sparse; it extracts the nonzeros from a sparse matrix.

3. `speye` creates a sparse identity matrix.

4. `spdiags` constructs a sparse matrix by specifying the values on diagonals.

5. `sprand` and `sprandn` construct random sparse matrices with specified density.

6. `spones` creates a sparse matrix with the same non-zero structure as a given sparse matrix, where all the non-zero values are 1.

### 2.4.2 Element-wise Matrix Arithmetic

Sparse matrix arithmetic is implemented using a sparse accumulator (SPA). Gilbert, Moler and Schreiber [56] discuss the design of the SPA in detail. Briefly, a SPA uses a dense vector as intermediate storage. The key to making a SPA work is to maintain auxiliary data structures that allow direct ordered access to only the non-zero elements in the SPA. Star-P uses a separate SPA for each processor.

### 2.4.3 Matrix multiplication

#### Sparse matrix dense vector multiplication

A sparse matrix can be multiplied by a dense vector either on the right or the left. The CSR data structure used in Star-P is efficient for multiplying a sparse matrix by a dense vector: $y = A * x$. It is efficient for communication and shows good cache behavior for the sequential part of the computation. Our choice of the CSR data structure was heavily influenced by our desire to have good matvec performance, since matvec forms the core computational kernel for many iterative methods.

The matrix $A$ and vector $x$ are distributed across processors by rows. The submatrix of $A$ on each processor will need some subset of $x$ depending upon its sparsity structure. When matvec is invoked for the first time on a dsparse matrix $A$, Star-P computes a communication schedule for $A$ and caches it. When later matvecs are performed using the same $A$, this communication schedule does not need to be recomputed, which saves some computing and communication overhead, at the cost of extra space required to save the schedule. We experimented with overlapping computation and communication in matvec. It turns out in many cases that this is less efficient than simply performing the communication first, followed by the computation. As computer architectures evolve, this decision may need to be revisited.

Communication in matvec can be reduced by graph partitioning. If fewer edges cross processors, less communication is required during matvec. STAR-P can use several of the available tools for graph partitioning [29, 60, 100]. However, STAR-P does not perform graph partitioning automatically during matvec. The philosophy behind this decision is similar to that in MATLAB: reorganizing data to make later operations more efficient should be possible for the user, but not automatic.

When multiplying from the left, $y = x' * A$, the communication is not as efficient. Instead of communicating the required subpieces of the source vector, each processor computes its own destination vector. All partial destination vectors are then summed up into the final destination vector. The communication required is always $O(n)$. The choice of the CSR data structure, while making the communication more efficient when multiplying from the right, makes it more difficult to multiply on the left.

Sparse matrix dense matrix multiplication in STAR-P is implemented as a series of matvecs. Such operations although not very common, do often show up in practice. It is tempting to simply convert the sparse matrix to a dense

```
function C = mult_inner_prod (A, B)
% Inner product formulation of matrix multiplication

for i = 1:n           % For each row of A
   for j = 1:n        % For each col of B
      C(i, j) = A(i, :) * B(:, j);
   end
end
```

**Figure 2.2:** Inner product formulation of matrix multiplication. Every element of $C$ is computed as a dot product of a row of $A$ and a column of $B$

matrix and perform dense matrix multiplication, since the result is dense anyways. Doing so, however, would require extra flops. Such a scheme may also be inefficient in storage if the resulting matrix is smaller in dimensions than the sparse argument.

### Sparse matrix sparse matrix multiplication

The multiplication of two sparse matrices is an important operation in Star-P. It is a common operation for operating on large graphs. Its application to graph manipulation is described in chapter 3, and in the implementation of a multigrid solver in chapter 5. The sparse matrix multiplication algorithm we use is discussed by Robertson [103].

The computation for matrix multiplication can be organized in several ways, leading to different formulations. One common formulation is the inner product

```
function C = mult_outer_prod (A, B)
% Outer product formulation of matrix multiplication

for k = 1:n
   C = C + A(:, k) * B(k, :);
end
```

**Figure 2.3:** Outer product formulation of matrix multiplication. $C$ is computed as a sum of $n$ rank one matrices.

formulation, as shown in code fragment 2.2. In this case, every element of the product $C_{ij}$ is computed as a dot product of a row $i$ in $A$ and a column $j$ in $B$.

Another forumulation of matrix multiplication is the outer product formulation (code fragment 2.3. The product is computed as a sum of $n$ rank one matrices. Each rank one matrix is computed as the outer product of column $k$ of $A$ and row $k$ of $B$.

Matlab stores its matrices in the CSC format. Clearly, computing inner products (code fragment 2.2) is inefficient, since rows of $A$ cannot be efficiently accessed without searching. Similarly, in the case of computing outer products (code fragment 2.3), rows of $B$ have to be extracted. The process of accumulating successive rank one updates is also inefficient, as the structure of the result changes with each successive update.

The computation can be setup so that $A$ and $B$ are accessed by columns, computing one column of the product $C$ at a time. Code fragment 2.4 shows

```
function C = mult_csc (A, B)
% Multiply matrices stored in compressed sparse column format

for i = 1:n
   for k where B(k,j) ~= 0
      C(:, j) = C(:, j) + A(:, k) * B(k, j);
   end
end
```

**Figure 2.4:** The column-wise formulation of matrix multiplication accesses all matrices$A$, $B$ and $C$ by columns only

how column $j$ of $C$ is computed as a linear combination of the columns of $A$ as specified by the nonzeros in column $j$ of $B$. Figure 2.5 shows the same concept graphically.

STAR-P stores its matrices in CSR form. As a result, the computation is setup so that only rows of $A$ and $B$ are accessed, producing a row of $C$ at a time. Each row $i$ of $C$ is computed as a linear combination of the rows of $B$ specified by non-zeros in row $i$ of $A$ (code fragment 2.6).

The performance of sparse matrix multiplication in parallel depends upon the non-zero structures of $A$ and $B$. A well-tuned implementation may use a polyalgorithm. Such a polyalgorithm may use different communication schemes for different matrices. For example, it may be efficient to broadcast the local part of a matrix to all processors, but in other cases, it may be efficient to send only the required rows. On large clusters, it may be efficient to interleave

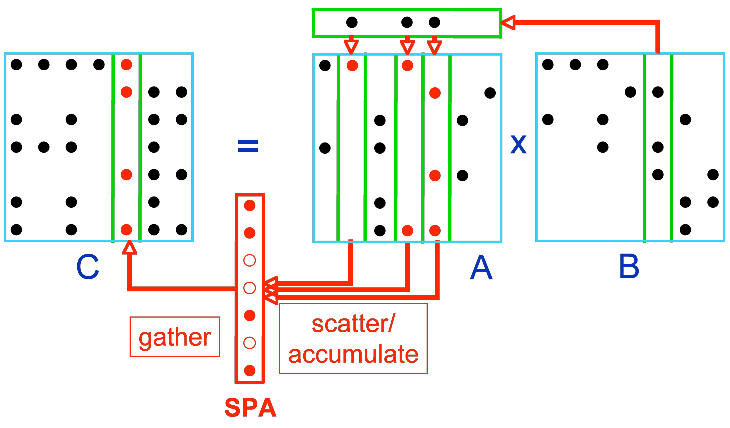


**Figure 2.5:** Multiplication of sparse matrices stored by columns. Columns of $A$ are accumulated as specified by the non-zero entries in a column of $B$ using a SPA. The contents of the SPA are stored in a column of $C$ once all required columns are accumulated. Image reproduced with permission from John Gilbert.

communication and computation. On shared memory architectures, however, most of the time is spent accumulating updates, rather than in communication. In such cases, it may be more efficient to schedule the communication before the computation. In the general case, the space required to store $C$ cannot be determined quickly, and Cohen's algorithm [34] may be used in such cases.

### 2.4.4 Sparse matrix indexing, assignment, and concatenation

Several choices are available to the implementor to design primitives upon which a sparse matrix library is built. One has to decide early on in the design

```
function C = mult_csr (A, B)
% Multiply matrices stored in compressed sparse row format

for i = 1:n
   for k where A(k,j) ~= 0
      C(i, :) = C(i, :) + A(i, k) * B(k, :);
   end
end
```

**Figure 2.6:** The row-wise formulation of matrix multiplication accesses all matrices $A$, $B$ and $C$ by rows only.

phase which operations will form the primitives and how other operations will be derived from them.

The syntax of matrix indexing in STAR-P is the same as in MATLAB. It is of the form $A(p, q)$, where $p$ and $q$ are vectors of indices.

```
>> B = A(p,q)
```

In this case, the indexing is done on the right side of "=", which specifies that $B$ is assigned a submatrix of $A$. This is the `subsref` operation in MATLAB.

```
>> B(p,q) = A
```

On the other hand, indexing on the left side of "=" specifies that $A$ should be stored as a submatrix of $B$. This is the `subsasgn` operation in MATLAB. Repeated indices in subsref cause replication of rows and columns. However, subsasgn with repeated indices is not well defined.

MATLAB supports horizontal and vertical concatenation of matrices. The following code, for example, concatenates $A$ and $B$ horizontally, $C$ and $D$ horizontally, and finally concatenates the results of these two operations vertically.

```
>> S = [ A B; C D ]
```

All of these operations are widely used, and users often do not give second thought to the way they use indexing operations. The operations have to accept any sparse matrix, and return a result in the same form with reasonable performance. In a parallel implementation such as STAR-P, another dimension of complexity is added by communication. Performance of sparse indexing operations depends upon the underlying data structure, the indexing scheme being used, the non-zero structure of the matrix, and the speed of the communication network.

Our implementation uses `sparse` and `find` as primitives to implement sparse indexing. The idea is actually quite simple. First, find all elements that match the selection criteria on each processor. Depending on the operation being performed, rows and columns may need to be renumbered. Once all processors have picked the non-zero tuples which will contribute to the result, call `sparse` to assemble the matrix.

Such a scheme is elegant because all the complexity of communication is hidden in the call to `sparse`. This simplifies the job for the implementor, who can focus a great deal of effort into developing an efficient `sparse` routine.

### 2.4.5 Sparse matrix transpose

Matrix transpose exchanges the rows and columns of all elements of the matrix. Transpose is an important operation, and has been widely studied in the dense case. In a sparse transpose, apart from communication, the communicated elements have to be re-inserted into the sparse data structure. The MATLAB syntax for matrix transpose is as follows:

```
>> S = A'
```

Sparse matrix transpose can be easily implemented using the sparse and find primitives. First, find all nonzero elements in the sparse matrix with `find`. Then construct the transpose with `sparse`, exchanging the vectors for rows and columns.

```
[I, J, V] = find (S);
St = sparse (J, I, V);
```

### 2.4.6 Direct solvers for sparse linear systems

MATLAB solves the linear system $Ax = b$ with the matrix division operator, $x = A\backslash b$. In sequential MATLAB, $A\backslash b$ is implemented as a polyalgorithm [56], where every test in the polyalgorithm is cheaper than the next one.

1. If $A$ is not square, solve the least squares problem.
2. Otherwise, if $A$ is triangular, perform a triangular solve.
3. Otherwise, test whether $A$ is a permutation of a triangular matrix (a "morally triangular" matrix), permute it, and solve it if so.
4. Otherwise, if $A$ is Hermitian and has positive real diagonal elements, find a symmetric approximate minimum degree ordering $p$ of $A$, and perform the Cholesky factorization of $A(p, p)$. If successful, finish with two sparse triangular solves.
5. Otherwise, find a column minimum degree order $p$, and perform the LU factorization of $A(:, p)$. Finish with two sparse triangular solves.

The current version of MATLAB uses CHOLMOD [31] in step 4, and UMFPACK [38] in step 5. MATLAB also uses LAPACK [7] band solvers for banded matrices in its current polyalgorithm.

Different issues arise in parallel polyalgorithms. For example, morally triangular matrices and symmetric matrices are harder to detect in parallel. In the next section, we present a probabilistic approach to test for matrix symmetry. Design of the best polyalgorithm for "backslash" in parallel is an active research problem. For now, Star-P offers the user a choice between two existing message-passing parallel sparse solvers: MUMPS [5] and SuperLU_Dist [86].

Sparse solvers are extremely complex pieces of software, often taking several years to develop. They use subtle techniques to extract locality and parallelism, and have complex data structures. Most sparse solvers provide an interface only to solve linear systems, $x = A\ b$. They often do not provide an interface for the user to obtain the factors from the solve, $[L, U] = lu(A)$. Matlab uses UMFPACK only when backslash or the four output version of `lu` is used, $[L, U, P, Q] = lu(A)$. Many Matlab codes store the results of LU factorization for later use. Since Star-P does not yet provide a sparse $lu$ implementation, it may not be able to run certain Matlab codes in parallel.

### 2.4.7 Iterative solvers for sparse linear systems

Iterative solvers for sparse linear systems include a wide variety of algorithms that use successive approximations at each step. *Stationary* methods are older, simpler, but usually not very effective. These include methods such as Jacobi, Gauss-Seidel and successive overrelaxation. *Nonstationary* methods, also known as Krylov subspace methods, are relatively modern, and based on the idea of sequences of orthogonal vectors. Their convergence typically depends upon the condition number of the matrix. Often, a *preconditioner* is used to transform a given matrix into one with a more favorable spectrum, accelerating convergence.

Iterative methods are not used by default for solving linear systems in STAR-P. This is mainly because efficient methods are not yet available for all classes of problems. *Conjugate Gradient* (CG) works well for matrices which are symmetric and positive definite (SPD). Methods such as *Generalized Minimal Residual* (GMRES) or *Biconjugate gradient* (BiCG, BiCGStab) are used for unsymmetric matrices. However, convergence may be irregular, and it is also possible that the methods may break.

Even when using CG, a preconditioner is required for fast convergence. Preconditioners are often problem specific; their construction often requires knowledge of the problem at hand. An exciting new area of research is combinatorial preconditioners. The graph toolbox in STAR-P provides tools for users to build such preconditioners. This is further described in chapters 3 and 5.

Although STAR-P does not use iterative methods by default, it provides several tools for users to use them when suitable. Preconditioned iterative methods from software such as Aztec [107] and Hypre [47] may also be used in STAR-P through the STAR-P SDK [75].

### 2.4.8 Eigenvalues and singular values

STAR-P provides eigensolvers for sparse matrices through PARPACK [90]. PARPACK uses a reverse communication interface, in which it provides the essential routines for the Arnoldi factorization, and requires the user to provide routines for matvec and linear solves. STAR-P implementations of matvec and linear solvers are discussed in earlier sections of this chapter.

STAR-P retains the same calling sequence as the MATLAB `eigs` function. STAR-P can also provide singular values in a similar fashion, and retains the same calling sequence for `svds`.

### 2.4.9 Visualization of sparse matrices

We have experimented with a few different methods to visualize sparse matrices in STAR-P. A MATLAB spy plot of a sparse matrix shows the positions of all non-zeros. For extremely large sparse matrices, this approach does not work very well since each pixel on the screen represents a fairly large part of a matrix. Figure 2.7 shows a MATLAB spy plot of a web crawl matrix. It also shows a coloured spy plot with a different color for each processor. The row-wise

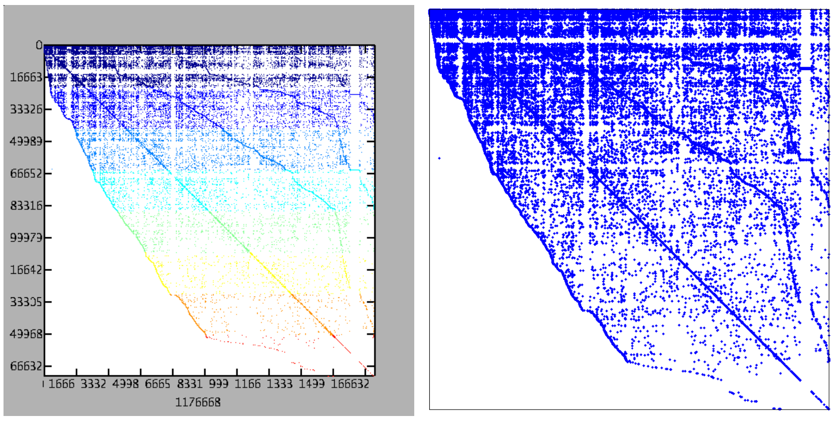


**Figure 2.7:** Matlab and Star-P spy plots of a web crawl sparse matrix. The Star-P plot also exposes the underlying block row distribution of the sparse matrix. The matrix was constructed by running a breadth-first web crawl from www.mit.edu.

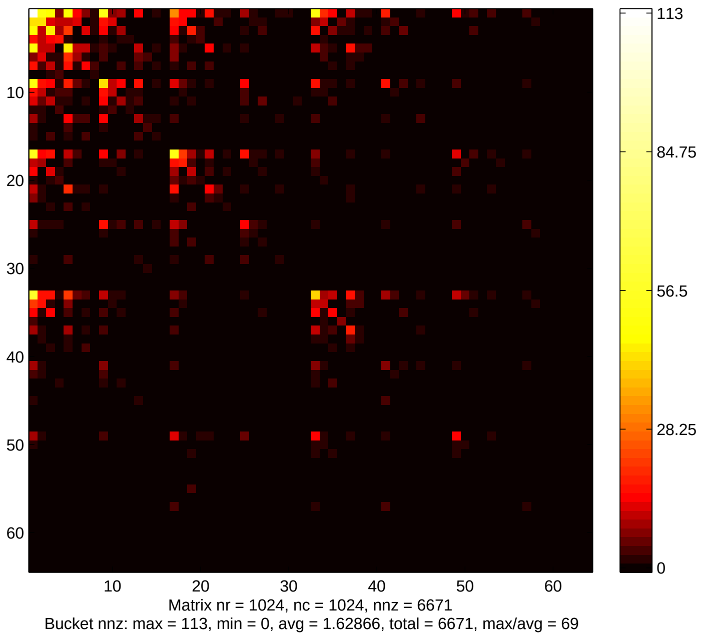


**Figure 2.8:** A density spy plot. For large matrices, spy may not display the underlying structure. A density plot colors each pixel according the density of the area of the sparse matrix it represents.

distribution of the matrix is clearly observed in the colored spy plot. Another approach is to use a 2D histogram or a density spy plot, such as the one in Figure 2.8, which uses different colors for different non-zero densities. `spyy` uses sparse matrix multiplication to produce density spy plots. It is similar to the `cspy` routine in CSparse [39], with the exception that `cspy` is implemented as in $C$, and cannot be used in STAR-P. `spyy` operates on large dsparse matrices on the backend, but the resulting images are small, which are easily communicated to the frontend.

Sparse matrix visualization is an exciting area with a lot of open questions. Since visualization is not our main contribution we do not pursue this topic further. We will briefly return to it later in chapter 5 in the context of graphs.

## 2.5 Looking forward: A next generation parallel sparse library

We discuss the design goals of a next generation parallel sparse library in this section.

Our initial goal was to develop a parallel sparse library for STAR-P similar to the one in MATLAB. We wanted it to be robust, scalable, efficient and simple. Hence, all the design decisions we made always favored robustness and simplicity. For instance, we decided early on to support only the CSR data structure to store sparse matrices, and use a 1D block layout by rows. A 2D layout may be more efficient, and it would definitely be nice to support other data structures for storage. However, these would complicate the implementation to a point where it may not be possible to implement all the operations we currently support for all combinations of data structures and layouts.

That said, there are some crucial differences between MATLAB and STAR-P's sparse libraries. First, MATLAB's focus is solely on numerical computing. However, sparse matrices are increasingly lending themselves to more than just numerical computing. Second, parallel computing is still not as easy as sequential computing. Parallel sparse matrices provide an elegant way for a user to represent large sparse datasets as matrices without using complex data structures to store and query them. Operations on such large datasets often require irregular communication and irregular data access. Sparse matrix computations allow this to be done in a concise and systematic way, without worrying about low level details. One example is the use of sparse matrices to represent graphs for combinatorial computing, as discussed further in chapter 3.

From our experience with the STAR-P sparse matrix library, we have learnt a few lessons which might be helpful for the development of the next generation parallel sparse library.

Although the CSR data structure has served us well, it does cause some problems. Matlab codes written to specifically take advantage of the column storage of sparse matrices must be rewritten to use row storage. Although it is not yet clear how much the performance difference may be for a real life application, it is inconvenient for a user writing highly tuned codes using sparse matrices. Such a code will also have different performance characteristics in Matlab and Star-P.

We propose adding a third design principle to the two stated in section 2.4. The difference in performance between accessing a sparse matrix by rows or columns must be minimal. Users writing sparse matrix codes should not have to worry about organizing their sparse matrix accesses by rows or columns, just as they do not worry about how dense matrices are stored.

For an example of the third principle, consider the operation below. It is much simpler to extract the submatrix specified by $p$ from a matrix stored in the CSR format than from a matrix stored in the CSC format. In the latter case, a binary search is required for every column, making the operation slower.

```
>> A = S(p, :)      % p is a vector
```

A parallel sparse library that needs to scale to hundreds or thousands of processors will not work well with a one-dimensional block layout. A two-dimensional block layout is essential for scalability of several sparse matrix op-

```
function B = index_by_mult (A, I, J)
% Index a matrix with matrix multiplication

[nrows, nccols] = size(A);
nI = length(I);
nJ = length(J);

% Multiply on the left to pick the required rows
row_select = sparse(1:nI, I, 1, nI, nr);

% Multiply on the right to pick the required columns
col_select = sparse(J, 1:nJ, 1, nc, nJ);

% Compute B with sparse matrix multiplication
B = row_select * A * col_select;
```

**Figure 2.9:** Matrix indexing and concatenation can be implemented using sparse matrix-matrix multiplication as a primitive. Multiplication from the left picks the necessary rows, while multiplication from the right picks the necessary columns.

erations. The benefits of 2D layouts are well known at this point, and we will not reiterate them. It is important to note that compressed row/column data structures are not efficient for storing sparse matrices in a 2D layout.

Another point of departure is a different primitive for indexing operations. Currently, we use the "sparse-find" method to perform all the indexing operations such as submatrix indexing, assignment, concatenation and transpose. We used the concept of the `sparse` function as a primitive using which we built the rest of the operations. We propose that a next generation sparse matrix library should use sparse matrix multiplication as the basic primitive in the library.

We illustrate the case of submatrix indexing using matrix multiplication. Suppose $A$ is a matrix, and we wish to extract the submatrix $B = A(I, J)$. Multiplying from the left picks out the rows, while multiplying from the right picks out the columns, as shown in code fragment 2.9.

We believe that it will be possible to implement sparse matrix multiplication more efficiently with a 2D block distribution than a 1D block distribution for large numbers of processors. Indexing operations may then be implemented using sparse matrix multiplication as a primitive. An efficient sparse matrix multiplication implementation might actually use a polyalgorithm to simplify the implementation for special cases when more information is available about the structure of matrices being multiplied, as is the case for indexing operations.

The choice of a suitable data structure to store sparse matrices in a 2D block layout and allow efficient matrix multiplication still remains an open question for future research. We believe that once the right data structure is selected, there will not be a large difference in performance when multiplying from the left or the right. This directly translates into symmetric performance for all indexing operations when accessing a matrix either by rows or columns. We believe this will lead to higher programmer productivity, freeing users from tuning their codes in specific ways that depend on knowledge of the underlying implementation of indexing operations.

## 2.6 Conclusion

The goal of sparse matrix support in STAR-P is to provide an interactive environment for users to perform operations on large sparse matrices in parallel, while being compatible with MATLAB. Our current implementation is complete, and has been used successfully for several problems. We also make recommendations for a next generation parallel sparse library which uses sparse matrix multiplication as its key primitive.

# Chapter 3

# Parallel Sparse Matrices and Graph Algorithms

## 3.1 Motivation

High performance applications increasingly combine numerical and combinatorial algorithms. Past research on high performance computation has focused mainly on numerical algorithms, and we have a rich variety of tools for high performance numerical computing. On the other hand, few tools exist for large scale combinatorial computing.

Our goal is to make it easy for scientists and engineers to develop applications using modern numerical and combinatorial methods with as little effort as possible. Sparse matrix computations allow structured representation of irregular data structures, decompositions, and irregular access patterns in parallel applications.

Sparse matrices are a convenient way to represent graphs. Since sparse matrices are first class citizens in MATLAB and many of its parallel dialects [32], it is natural to use the duality between sparse matrices and graphs to develop a unified infrastructure for numerical and combinatorial computing.

Several researchers are building a library of parallel graph algorithms: the Parallel Boost Graph Library (PBGL) [65] at Indiana University, another one at Georgia Tech [11], and the Multi-threaded Graph Library (MTGL) [49] at Sandia National Laboratory. The PBGL builds upon the Boost graph library [111], and uses MPI for parallelism. Both the Georgia Tech library and MTGL focus heavily on thread-level parallelism.

Our approach relies upon representing graphs with sparse matrices. We use the distributed sparse array infrastructure in STAR-P to build an infrastructure for computing with graphs. Parallelism is derived from operations on parallel sparse matrices. This approach results in several desirable characteristics. First, the implementation is completely written in a high-level language such as MATLAB making the codes are short, simple, and readable. Our implementation uses a single thread of control (SIMD), making it easier to write and debug programs. The distributed sparse array implementation in STAR-P provides a set of well-tested primitives with which graph algorithms can be built. The effi-

| Sparse matrix operation | Graph operation |
|---|---|
| `G = sparse (U, V, W)` | Construct a graph from an edge list |
| `[U, V, W] = find (G)` | Obtain the edge list from a graph |
| `vtxdeg = sum (spones(G))` | Node degrees for an undirected graph |
| `indeg = sum (spones(G))` | Indegrees for a directed graph |
| `outdeg = sum (spones(G), 2)` | Outdegrees for a directed graph |
| `N = G(i, :)` | Find all neighbors of node $i$ |
| `Gsub = G(subset, subset)` | Extract a subgraph of G |
| `G(i, j) = W` | Add or modify graph edges |
| `G(i, j) = 0` | Delete a graph edges |
| `G(I, I) = []` | Remove nodes from a graph |
| `G = G(perm, perm)` | Permute nodes of a graph |
| `reach = G * start` | Breadth-first search step |

**Table 3.1:** There exists a correspondence between sparse matrix operations and graph operations. Many simple sparse matrix operations can be used to directly perform basic operations on graphs.

ciency of our graph algorithms depends upon efficiency of the underlying sparse matrix infrastructure.

The primitives described in the next section are used to implement several graph algorithms in the "Graph Algorithms and Pattern Discovery" toolbox (GAPDT) developed by Gilbert, Reinhardt and Shah [58, 101]. The toolbox was designed from the outset to run interactively with terascale graphs via STAR-P. Many of the components provided scale to tens/hundreds of processors. High performance and interactivity are salient features of this toolbox.

## 3.2 Sparse matrices and graphs

Every sparse matrix problem is a graph problem, and every graph problem is a sparse matrix problem. We discuss some of the basic principles in the design of a comprehensive infrastructure for sparse matrix data structures and algorithms in chapter 2. The same principles apply to efficient operations on large sparse graphs.

1. Storage for a sparse matrix should be $\Theta(\max(n, nnz))$
2. An operation on sparse matrices should take time approximately proportional to the size of the data accessed and the number of nonzero arithmetic operations on it.

A graph consists of a set of nodes $V$, connected by edges $E$. A graph can be specified by tuples $(u, v, w)$ indicating a directed edge of weight $w$ from node $u$ to node $v$. This is the same as a nonzero $w$ at location $(u, v)$ in a sparse matrix. Following principle 1, the storage required is $\Theta(|V| + |E|)$. An undirected graph is represented by a symmetric sparse matrix. A correspondence between sparse matrix operations and graph operations is listed in Table 3.1. The basic design principles silently come into play in all cases.

### 3.2.1 Sparse matrix multiplication

We argue in chapter 2 that sparse matrix multiplication can be a basic building block for sparse matrix and graph based computations. There is a correspondence between path problems on graphs and operations on sparse matrices [4, 117]. Specifically, sparse matrix multiplication on semirings can be used to implement a wide variety of graph algorithms.

**Definition**: A *closed semiring* is a system $(S, \oplus, \otimes, 0, 1)$, where $S$ is a set of elements over which the binary operations of addition and multiplication are defined. $\oplus$ denotes addition; $\otimes$ denotes multiplication. $S$ has the following properties:

1. $(S, \oplus, 0)$ is a commutative monoid with 0 as its identity element.
   - $\oplus$ is associative: $a \oplus (b \oplus c) = (a \oplus b) \oplus c$.
   - $\oplus$ is commutative: $a \oplus b = b \oplus a$.
   - $\oplus$ has 0 as its identity element: $a \oplus 0 = 0 \oplus a = a$.
2. $(S, \otimes, 1)$ is a monoid with 1 as its identity element.
   - $\otimes$ is associative: $a \otimes (b \otimes c) = (a \otimes b) \otimes c$.
   - $\otimes$ has 1 as its identity element: $a \otimes 1 = 1 \otimes a = a$
3. $\otimes$ distributes over $\oplus$.

- Left distributivity: $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$.
- Right distributivity: $(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$.

4. 0 is an annihilator for all elements in $R$.

   - $a \otimes 0 = 0 \otimes a = 0$.

It is straightforward to extend a given matrix multiplication code to allow operations over arbitrary semirings. Modern object oriented languages include facilities for operator overloading that allow $+$ and $\times$ to be overloaded for a given semiring. We extended STAR-P's sparse matrix multiplication to allow multiplication over semirings.

Some algorithms that can be expressed in this framework are breadth-first search, transitive closure [124], the Floyd-Warshall algorithm [50] for computing all pairs shortest paths, and Cohen's algorithm [34] for estimating the fill resulting from sparse matrix multiplication. Examples of semirings useful for graph algorithms are as follows:

- $(\mathbb{R}, +, \times, 0, 1)$ is the most commonly used semiring. It is the semiring used by default.
- $(\{0, 1\}, |, \&, 0, 1)$ is a semiring using boolean operators. It is useful in search algorithms such as breadth-first search.

- $(\mathbb{R}, MIN, \otimes, +\infty, 1)$ is an example of a semiring useful for contracting nodes in a graph. $\otimes$ only allows multiplication with 1. In this specific example, minimum weight edges are retained between contracted nodes in the graph.
- The $(\mathbb{R}, ARGMIN, \otimes, +\infty, 1)$ semiring can be used instead, if an algorithm requires edge information (such as the node contributing the edge) rather than the weight of the edge.
- The $(\mathbb{R}, MIN, \otimes, +\infty, 1)$ semiring can also be used to implement Cohen's algorithm to predict the number of nonzeros in the rows or columns of the result of sparse matrix multiplication.
- The $(\mathbb{R}, MIN, +, +\infty, 0)$ semiring can be used to implement single source shortest path algorithms such as Dijkstra's algorithm, and all pairs shortest path algorithms such as Floyd-Warshall.

## 3.3 Graph algorithms

We present a few commonly used operations on graphs in combinatorial scientific computing. The list, though by no means comprehensive, includes several graph algorithms: breadth-first search, connected components, strongly

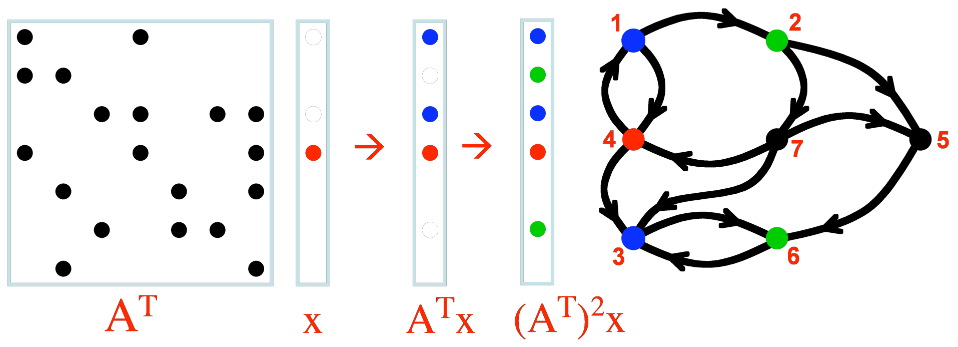


**Figure 3.1:** Breadth-first search on a graph is implemented with matrix vector multiplication. Initialize a sparse vector with a 1 in the position corresponding to the start node. Repeated multiplication yields multiple breadth-first steps on the graph. Note that $A$ can be either symmetric or unsymmetric. Image reproduced with permission from John Gilbert.

connected components, maximal independent set, maximum weight spanning tree, and graph contraction.

### 3.3.1 Breadth-first search

Consider breadth-first search (BFS). A BFS can be performed by multiplying a sparse matrix $G$ with a sparse vector $x$. Consider the simplest case of performing a BFS starting from node $i$. In this case, we set $x(i) = 1$, all other elements being zeros. $y = G * x$ simply picks out column $i$ of $G$, which contains the neighbors of node $i$. If this step is repeated, the multiplication will result in a vector that is a linear combination of all columns of $G$ corresponding to the nonzero elements in vector $x$, or all nodes that are up to 2 hops away from node $i$. Figure 3.1 shows the correspondence between breadth-first search on a directed graph and matrix vector multiplication.

We can also perform several independent BFS searches simultaneously by using sparse matrix matrix multiplication (described in chapter 2). Instead of multiplying with a vector, we multiply with a matrix, with one nonzero in each column at some row $i$, where $i$ is the starting node. So, we have $Y = G{*}X$, where column $j$ of $Y$ contains the result of performing an independent BFS starting from the node (or set of nodes) specified in column $j$ of $X$. The resulting time complexity for performing BFS with operations on sparse matrices is the same as that obtained by performing operations on other efficient graph data structures.

### 3.3.2 Connected Components

A *connected component* in an undirected graph is a maximal connected subgraph. Every node in the graph belongs to exactly one connected component.

We implement the Awerbuch-Shiloach [8] algorithm to find connected components of a graph in parallel (code fragment 3.2). The algorithm is very similar to the original Shiloach-Vishkin [110], but simpler. The algorithm is based on combining trees of nodes, such that all nodes in a given tree belong to the same connected component. The roots of the tree serve as labels for the trees. The

```
function D = components (G)
% Connected components of a graph

D = 1:length(G);    % Parent information
[u, v] = find(G);

while true
  % Conditional hooking
  Du = D(u);    % Parents of nodes u
  Dv = D(v);    % Parents of nodes v

  % Locate trees to be hooked.
  hook = find (Du == D(Du) & Dv < Du);
  Du = Du(hook);
  Dv = Dv(hook);
  D(Du) = Dv;

  % Check for termination
  if nnz(check_stars(D)) == n; break; end

  % Unconditional hooking
  D = unconditional_hooking (D, star, u, v);

  % Pointer jumping
  while D(D) ~= D;  D = D(D);  end
end
```

**Figure 3.2:** The Awerbuch-Shiloach connected components algorithm. The algorithm works by grafting rooted trees onto other trees. The Star-P implementation manipulates only dense vectors.

algorithm returns a label for each node of the graph, corresponding to the root of the tree it belongs to.

The trees are represented by a parent information vector. The parent of node $i$ is stored in $D(i)$. The two basic operations used to find connected components are hooking and shortcutting.

Two different types of hooking steps are used. Conditional hooking combines two trees so that the larger numbered root is below the smaller. Unconditional hooking only hooks trees that are not hooked by conditional hooking. This is mainly to avoid worst case scenarios. Shortcutting (or pointer jumping) simply flattens the trees so that all trees are of height 1. $O(\log n)$ iterations of hooking and shortcutting are required to compute the connected components of a graph.

### 3.3.3 Maximal Independent Set

An *independent set* in a graph is a set of nodes, no two of which are adjacent. A maximal independent set is an independent set that is not properly contained in any independent set.

We use Luby's randomized algorithm [87] to compute a maximal independent set of a graph. The main body of our implementation is presented in code fragment 3.3. The algorithm is one of the simplest and most elegant parallel graph algorithms. It starts by selecting nodes in the graph with probability inversely proportional to their degrees (lines 1–6). If neighboring nodes are selected, it keeps those with higher degrees. These nodes are added to the maximal independent set. The algorithm then iterates on the subgraph that remains after removing the selected nodes and their neighbors (lines 20–22).

We briefly summarize Luby's key result, which makes this algorithm work. Let $m_1$ be the number of edges in the graph before the loop iteration, $m_2$ be the number of edges after the loop iteration and $m_3$ be the number of edges eliminated. Then $m_1 = m_2 + m_3$. The expected number of edges eliminated in any iteration is strictly greater than $m_1/8$. Clearly, the number of loop iterations required is $O(\log n)$ with very high probability.

### 3.3.4 Maximum weight spanning trees

Given an undirected weighted graph $G$, a spanning tree contains a maximal set of edges of $G$ that contains no cycles. A maximum weight spanning tree has the maximum weight of all possible spanning trees of $G$.

**Theorem 3.3.1.** *[96] Let $G(V, E)$ be an undirected weighted graph. Let $U$ be any subset of the nodes $V$ of $G$. Then the maximum weight edge linking a node in $U$ to a node in $V - U$ is in some maximum weight spanning tree of $G$.*

Code fragment 3.4 describes the computation of the maximum weight spanning tree (MWST). Boruvka's theorem (theorem 3.3.1) [96] says that every

```
function IS = mis (G)
% Maximal independent set of a graph

IS = [];
while length(G) > 0
  % Select vertices with probability 1/(2*degree)
  degree = sum (G,2);
  prob = 1 ./ (2 * degree);
  select = rand (length(G), 1) <= prob;

  % If neighbors selected, keep nodes with higher degree
  neighbors = select & G * select;
  some_neighbors = ...textcolorcomment; % lower degree nodes
  if ~isempty(neighbors); select(some_neighbors) = 0;  end

  % Add selected nodes to independent set
  IS = [IS find(select)];

  % Exclude neighbors of selected vertices
  remain = not (select | G * select);

  % Iterate on the remaining subgraph
  G = G (remain, remain);
end
```

**Figure 3.3:** Luby's algorithm is used for a parallel implementation of maximal independent sets. The algorithm proceeds by randomly selecting high degree nodes belonging to the MIS with a high probability. Neighbors of the selected nodes are ignored and the computation proceeds on the remaining subgraph. Some lines in this code are omitted for simplicity.

```
function G_mst = mst (G)
% Maximum weight spanning tree of a graph

SN = 1:length(G);
while max(SN) > 1
  % Contract graph into supernodes
  [G_SN E_SN] = contract (G, SN, 'max');
  G_SN = G_SN - diag(diag(G_SN));   % Ignore diagonal

  % Pick max weight edges for each node from G_SN
  [ign, U] = max (G_SN, [], 2);

  % Pick edges {U_mst, V_mst} specified by E_SN from G_SN
  T = sparse ([U_mst; V_mst], [V_mst; U_mst], 1, n, n);
  G_mst = G_mst | T;

  % Use pointer jumping to find supernodes
  SN = components(G_mst);
end
```

**Figure 3.4:** Parallel maximum weight spanning tree. Boruvka's algorithm picks the heavy edges incident on a node (or a supernode), which always belong to the MWST. The selected edges define new supernodes based on connectivity, and the algorithm iterates on the contracted subgraph. This algorithm is also referred to as Sollin's algorithm. Some lines of this code are omitted for simplicity.

maximum weight edge incident on a node belongs to the MWST, assuming all edge weights are unique. As a result, at least $N/2$ edges of the MWST are selected in an iteration (line 10). There may be up to $N/2$ connected components (supernodes) after the first iteration, since the graph of the spanning tree may not necessarily be connected. These connected components are determined efficiently by pointer jumping (line 20). The algorithm then proceeds recursively on the contracted subgraph, requiring $O(\log N)$ iterations to construct the MWST.

### 3.3.5 Strongly connected components

A strongly connected component of a directed graph is a maximal subset of nodes containing a directed path from every node to all other nodes in the subset. We implement the Divide and Conquer Strongly Connected Components (DCSC) algorithm proposed by Fleischer, Hendrickson and Pinar [49] to compute the strongly connected components of an undirected graph.

A node $v$ is reachable from a node $u$ if there is a sequence of directed edges from $u$ to $v$. $Desc(G, v)$, the descendants of $v$, is the set of subset of nodes in $G$ which are reachable from $v$. Similarly, $Pred(G, v)$, the predecessors of $v$, is the subset of nodes from which $v$ is reachable. The descendants of $v$ are found by finding the predecessors of $v$ in the transpose of the input graph

```
function x = predecessor (G, v)
% Predecessors of a node in a graph

x = sparse (length(G), 1);
xold = x;
x(v) = 1;            % Start BFS from v.

while x ~= xold
    xold = x;
    x = x | G * x;  % Use matvec to find neighbors.
end
```

**Figure 3.5:** Predecessor computation can be achieved simply by breadth-first search (matrix vector multiplication). Descendants can then be computed as descendants in $G^T$ — the graph obtained by reversing the direction of the edges in $G$

(code fragment 3.5). A strongly connected component in $G$ containing node $v$ is denoted as $SCC(G, v)$.

**Theorem 3.3.2.** *[49] Let $G = (V, E)$ be a directed graph, with $v \in V$ a node in $G$. Then*

$$Desc(G, v) \cap Pred(G, v) = SCC(G, v)$$

**Theorem 3.3.3.** *[49] Let $G$ be a graph with node $v$. Every strongly connected component of $G$ is a subset of $Desc(G, v)$, a subset of $Pred(G, v)$ or a subset of $Rem(G, v)$.*

```
function scomponents (G, map)
% Strongly connected components of a graph

global label, count;

if isempty(G); return; end        % Recursion termination
v = 1 + fix(rand * n);            % Select a random pivot
pred = predecessor (G, v);        % Predecessors
desc = predecessor (G’, v);       % Descendants

% Intersection of predecessors and descendants is a SCC.
scc = pred & desc;
count = count + 1;
label(map(find(scc))) = count;

% Iterate over subgraph of predecessors
remain = find (xor (pred, scc));
scomponents (G(remain, remain), map(remain));

% Iterate over subgraph of descendants
remain = find (xor (desc, scc));
scomponents (G(remain, remain), map(remain));

% Iterate over remaining subgraph
remain = find (not (pred | desc));
scomponents (G(remain, remain), map(remain));
```

**Figure 3.6:** The Fleischer-Hendrickson-Pinar algorithm is used to compute the strongly connected components of a graph. Parallelism in this algorithm is achieved from matrix vector multiplication. It may also be achieved through divide-and-conquer. We extract parallelism by data parallel operations on large graphs. The computation can switch to sequential computation when the subproblems are small enough.

The algorithm to find strongly connected components of a graph (code fragment 3.6) follows from theorem 3.3.2. First, select a random pivot node $v$ (line 6). The intersection of its predecessors and descendants is a strongly connected component of $G$ (lines 13–15). Theorem 3.3.3 implies that the remaining nodes of $G$ are divided into three sets: $Desc(G, v)$, $Pred(G, v)$, and $Rem(G, v)$. Any additional strongly connected component must be entirely contained within one of these three sets. This leads to a divide and conquer formulation. An efficient implementation can use divide and conquer until the subproblems are small enough for sequential algorithms.

### 3.3.6 Graph contraction

Graph contraction is a common graph operation. Several graph algorithms proceed by solving the problem iteratively on smaller subgraphs. Often, nodes in a graph are renumbered during a computation. Graph contraction involves combining nodes in the graph, and edges incident on those nodes as well. As shown in code fragment 3.7, graph contraction is implemented with sparse matrix multiplication over semirings (lines 12–13). Any operation supported for the $\oplus$ operation on semirings can be used to combine edge weights during graph contraction. *argmin* and *argmax* are also supported as contraction operators. As a result, in cases when the *min* or *max* weight edge is selected, the routine

```
function [C, ARG] = contract (G, labels, op)
% Contract nodes of a graph

n = length (G);
m = max(labels);
S = sparse (labels, 1:n, 1, m, n);

if strcmp (op, 'plus')
  % C = S * G * S';
  [I J V] = find (G);
  C = sparse (labels(I), labels(J), V, m, m);
else
  C1 = mtimes_semiring (S, G, op, 'mult');
  [C, ARG] = mtimes_semiring (C1, S', op, 'mult');
end
```

**Figure 3.7:** Parallel graph contraction. Graph contraction can be performed with sparse matrix multiplication. It can also be implemented by using `sparse`, since `sparse` accumulates duplicate edges.

can also return the node upon which the edge is incident in the uncontracted graph.

## 3.4 Graph generators

We provide a few graph generators in our toolbox, since algorithm developers may not always have access to the graphs they are developing their algorithms for. We provide two random graph generators: an Erdős-Rényi generator [46], and the Recursive Matrix (R-MAT) generator [85]. Grid generators for grids arising from numerical solutions to PDEs are also included. All these graph

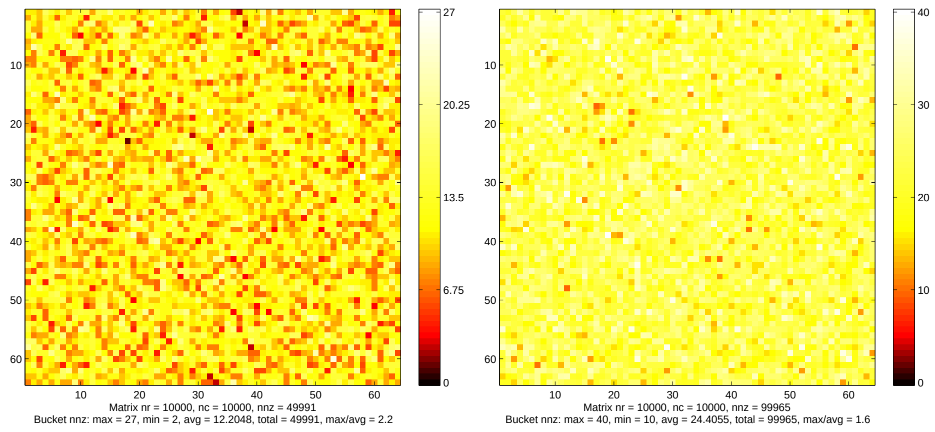


**Figure 3.8:** Density spy plots of random graphs. The image on the left is of a directed graph; the right image corresponds to a symmetric version of the same graph. The density scales are different on the two spy plots.

generators are fast and scalable, taking a parameter for scale as input. We have generated graphs with hundreds of millions of nodes using these graph generators.

## 3.4.1 Random graphs

In the Erdős-Rényi random graph model [46], each edge in the graph is picked independently with some fixed probability $p$. The MATLAB routines `sprand` and `sprandn` produce random sparse matrices with a specified density. These routines are included with STAR-P, and they produce distributed Erdős-Rényi random graphs.

The `sprand` and `sprandn` functions provide a quick way to generate random graphs to test graph algorithms during the design and testing phase. The ma-

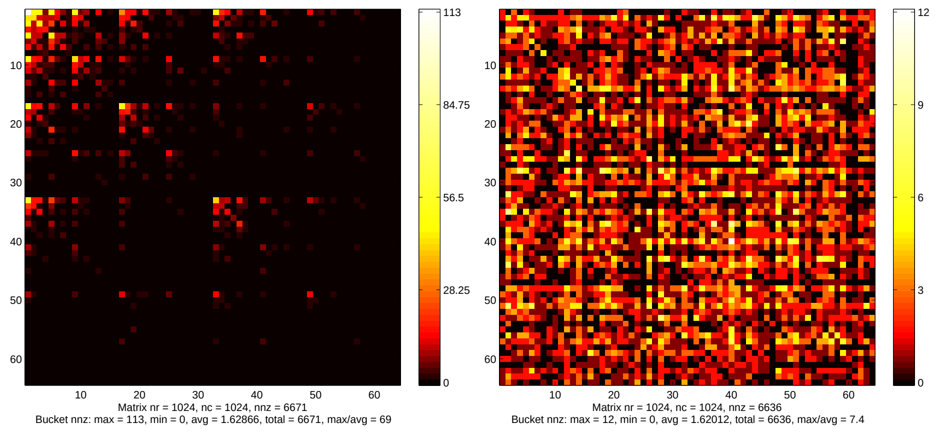


**Figure 3.9:** Density spy plots of a graph generated with the recursive matrix generator (R-MAT). The image on the left shows the recursive structure of the R-MAT graph. The image on the right shows the same graph with a randomized labeling of the nodes.

trices they generate are unsymmetric, resulting in directed graphs. Undirected graphs can be achieved by making the matrix symmetric by adding the matrix to its transpose.

### 3.4.2 Recursive matrix generator

R-MAT is another random graph model. It is defined by four parameters: $a$, $b$, $c$, and $d$, which sum to one. The adjacency matrix is recursively subdivided into four equal-sized partitions. Placing a nonzero in one of these partitions in the matrix corresponds to adding an edge in the graph. Edges are placed within these partitions with unequal probabilities. The algorithm generates the data tuples with a high degree of locality.

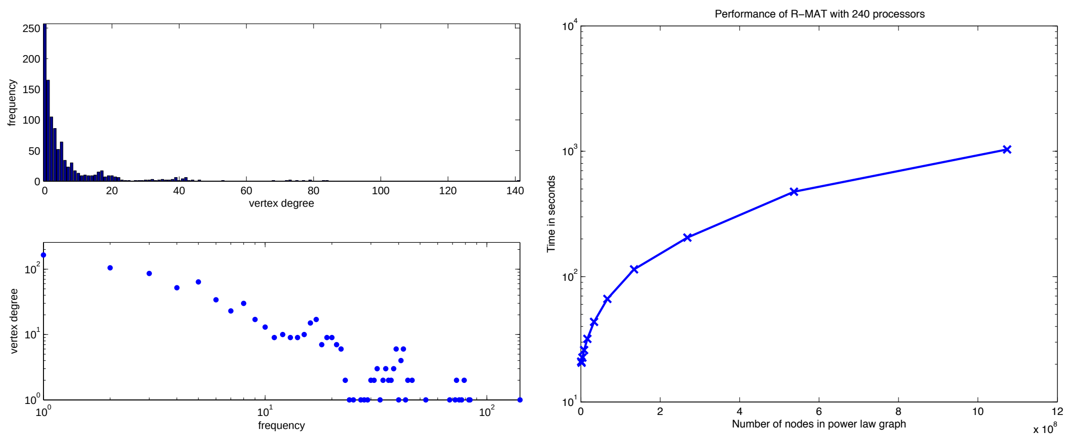


**Figure 3.10:** The degree distribution of nodes in the R-MAT graph are shown on the left. The right image shows the scaling of the R-MAT generator in Star-Pto large problems with 240 processors on a shared memory SGI Altix.

Kepner [10] provides a vectorized sequential Matlab implementation of R-MAT as part of the SSCA#2 reference implementation. The code runs efficiently in Star-P without modification. The default parameters produce power law graphs, which are sparse, scale-free, and tend to be good models of many real life networks. We adopt this code as the data generator for power law graphs in our toolbox.

Figure 3.10 shows a density spy plot of an R-MAT graph. The recursive structure is very clearly visible in the non-randomized plot. Locality and structure are destroyed when the nodes are relabeled randomly. Figure 3.10 shows the degree distribution of the generated power law graphs. The performance plot shows that the R-MAT generator scales well, all the way to a billion nodes[1].

[1]These experiments were performed using 240 processors of a 256 processor SGI Altix

```
function A = blockdiags(B, d, m, n)
% Sparse matrix formed from block diagonals

p = ncols (B) / length (d);
A = sparse (m*p, n*p);

for i=1:length(d)
  S = spdiags (ones(m,1), d(i), m, n);
  block = B(:, [(i-1)*p+1:i*p]);
  A = A + kron (S, block);
end
```

**Figure 3.11:** `blockdiags` is a generalization of the MATLAB `spdiags` call. It allows the user to construct a block diagonal matrix, where the blocks may themselves be sparse. The STAR-P implementation uses `kron` for speed.

### 3.4.3 Regular 2D and 3D grids

The GAPDT toolbox includes graph generators for 2D and 3D grids from the Meshpart toolbox [60], which correspond to the discretization of Laplace's equation on regular geometries. For example, we provide `grid5` (figure 3.13), `grid7` and `grid9`, which generate 5-point, 7-point and 9-point finite difference meshes on the unit square. `gridt` generates a finite difference mesh on an equilateral triangle, `grid3d` (figure 3.14) generates a 3D 7-point finite difference mesh, and `grid3dt` produces a 3D tetrahedral finite element mesh.

The grid generators use a generalization of the MATLAB `spdiags` routine — `blockdiags` (code fragment 3.11). `spdiags` constructs a sparse matrix with a specified diagonal structure. `blockdiags` generalizes this to construct a

```
function G = grid3d (k)
% Generate a 3-dimensional 7-point finite difference mesh

% Diagonals +1/-1
a = blockdiags ([-1 6 -1], -1:1, k, k);

% Diagonals +k/-k
I = speye (k, k);
aa = blockdiags ([-I a -I], -1:1, k, k);

% Diagonals +k^2/-k^2
II = speye(k^2,k^2);
G = blockdiags ([-II aa -II], -1:1, k, k);
```

**Figure 3.12:** Generate a grid for the 3D model problem. This illustrates the use of the `blockdiags` syntax to construct regular grids. The grid3d code is adapted from the Meshpart toolbox [60].

sparse matrix with a specified block-diagonal structure. The STAR-P version of `blockdiags` uses `kron` (Kronecker product) to generate a block diagonal matrix. Interestingly, parallel sparse `kron` is identical to sequential sparse `kron`, simply using dense matrix indexing and `sparse`. Once `blockdiags` is available, other grid generators are easily implemented. For example, code fragment 3.12 shows the construction of a regular 3D 7-point finite difference mesh.

## 3.5 Graph partitioning

The GAPDT toolbox includes a geometric partitioner [55] and a spectral partitioner [29]. The codes originate from the Meshpart toolbox [60]. Graph

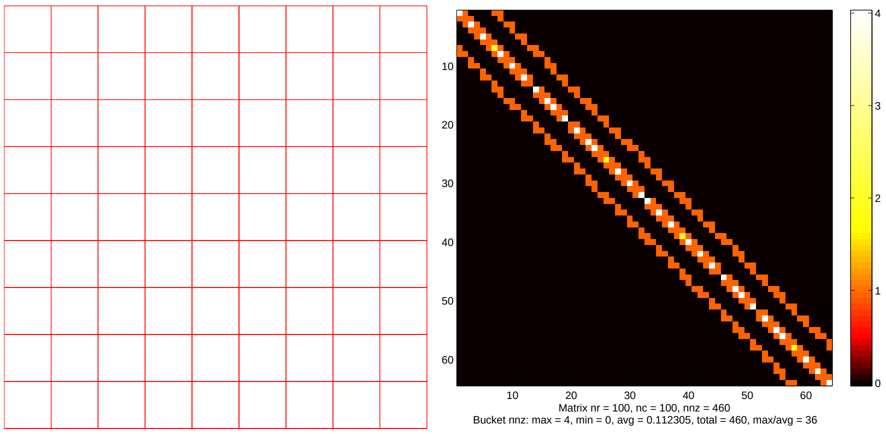


**Figure 3.13:** The model problem in 2D. The image to the left shows the graph; the image to the right shows a density spy plot of the graph.

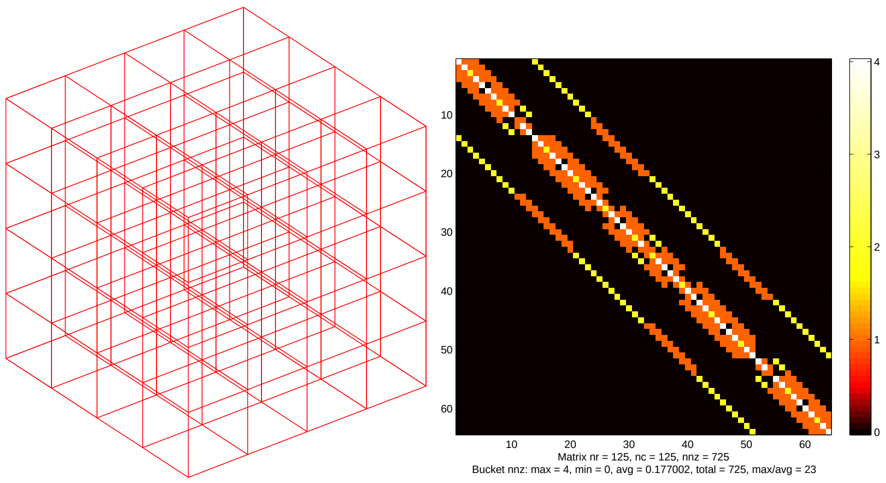


**Figure 3.14:** The model problem in 3D. The image to the left shows the graph; the image to the right shows a density spy plot of the graph.

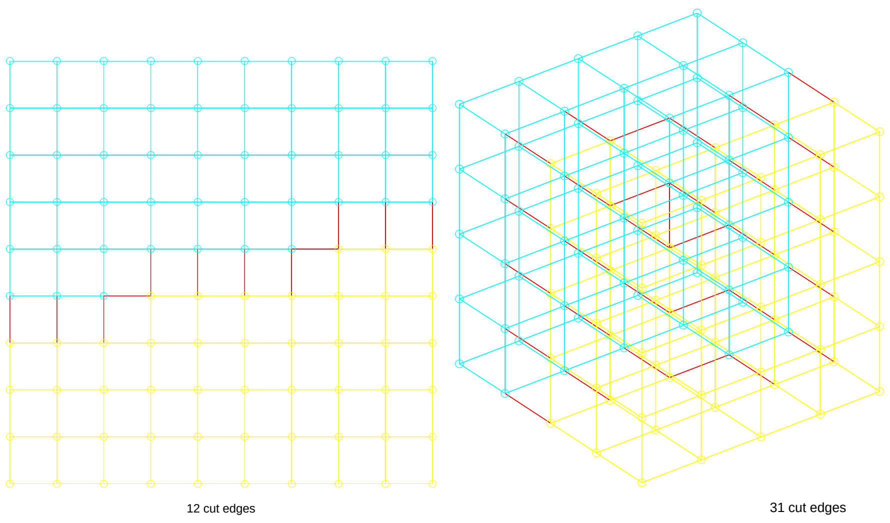


**Figure 3.15:** Geometric mesh partitioning works well on meshes arising from finite element discretizations. Geometric separators for the 2D and 3D model problems are shown.

partitioning is useful in a variety of applications such as solution of large sparse linear systems, VLSI circuit layout, and load balancing for parallel and distributed computing.

### 3.5.1 Geometric mesh partitioning

Graphs from large scale scientific problems are often defined geometrically. This method implements a geometric mesh partitioner based on the work of Miller, Teng, Thurston and Vavasis [94]. The method partitions a d-dimensional mesh by finding a suitable sphere in d-space, and dividing the nodes into those interior and exterior to the sphere. The cutting sphere is found by a randomized algorithm that involves a conformal mapping of the points on the surface of a sphere in $d+1$ space. Gilbert, Miller and Teng [55] report that their MATLAB implementation produces partitions that are better than the theoretical guarantees and competitive with those produced by other methods.

The algorithm parallelizes naturally. The original MATLAB code from Meshpart works with just one small modification in STAR-P to allow it to perform computations on small matrices on the frontend. The geometric mesh partitioner can partition regular grids with millions of nodes in parallel in a few minutes. For example, a 3D grid on the unit cube (figure 3.15) with 64 million nodes can be partitioned in 12 minutes with 14 processors on a shared memory Opteron system.

### 3.5.2 Spectral partitioning

Spectral partitioning originates from the ideas of Fiedler [48]. Pothen, Simon, and Liou suggested its application to graph partitioning in numerical computation [100]. Consider a graph $G$ with $n$ nodes. Let $L$ be the Laplacian matrix of $G$. It is constructed by making all off-diagonal elements negative. The diagonal is then adjusted so that row sums are zero. A *cut vector* $x$ is defined as a vector containing only +1 and -1 as its entries, where the sign determines which partition a node is in. The cut size, then, is four times $x^T Lx$. Therefore, the optimal graph bisection solves a discrete optimization problem:

$$\min_x x^T L x$$

$$x_i = \pm 1$$

$$\sum_i x_i = 0$$

Spectral bisection solves the continuous relaxation of this problem, rounding the continuous solution to a nearby discrete solution. It minimizes the Rayleigh quotient $x^T Lx/x^T x$ over all vectors $x$ orthogonal to the vector of all ones. The vector of all ones is an eigenvector corresponding to zero, the smallest eigenvalue of $L$.

$$\min_x x^T L x$$

$$||x|| = 1$$

$$\sum_i x_i = 0$$

The vector $x$ that solves this optimization problem is the so-called Fiedler vector of $L$, the eigenvector corresponding to the smallest non-zero eigenvalue. Nodes are partitioned around the median entry in the Fiedler vector.

Spectral partitioning in Star-P (code fragment 3.16) uses the `eigs` function to obtain the Fiedler vector [48] of the graph Laplacian. Since an interior eigenvalue is desired, a shift-and-invert method must be used, requiring the

```
function [p1, p2] = specpart (G)
% Spectral graph partitioning

% The Fiedler vector corresponds to the smallest
% non zero eigenvalue of the graph Laplacian
L = laplacian (G);
[V, D] = eigs (L, 2, 'SM');
fiedler = V(:, 1);

% Partition around the median of the Fiedler vector
m = median (fiedler);
p1 = find (fiedler < m);
p2 = find (fiedler >= m);
```

**Figure 3.16:** Spectral partitioning uses the Fiedler vector of the graph Laplacian to partition the graph. The Fiedler vector is the eigenvector corresponding to the smallest non zero eigenvalue of the graph Laplacian.

solution of a sparse linear system. This approach may be unsuitable for very large graphs.

## 3.6 Conclusion

We show the duality between sparse matrix algorithms and graph algorithms. This correspondence allows graph algorithms to be implemented as sparse matrix algorithms. We describe implementations of several parallel graph algorithms using the sparse matrix infrastructure in Star-P. Our implementation is packaged as the "Graph Algorithms and Pattern Discovery Toolbox" for

Matlab and Star-P. It provides a general infrastructure for users to develop their own highly scalable graph algorithms with relative ease.

# Chapter 4

# Parallel Sorting

## 4.1 Introduction

Traditionally, the field of scientific computing has been dominated by numerical methods. However, modern scientific codes often combine numerical methods with combinatorial methods. Sorting, a widely studied problem in computer science, is an important primitive for combinatorial scientific computing. As high performance computers become more affordable due to multi-core CPUs and commodity clustering, more and more scientific codes are written for parallel computers. Scientific programming environments such as MATLAB and STAR-P provide sorting as a built-in function. Parallel sorting can also form a basic building block to implement higher level combinatorial algorithms and computations with irregular communication patterns and workloads, such as parallel sparse matrix computations described in chapters 2 and 3.

We describe the design and implementation of an algorithm for parallel sorting on contemporary architectures. Distributed memory architectures are widely in use today. The cost of communication is several orders of magnitude larger than the cost of computation on such architectures. Often, it is not enough to tune existing algorithms. Newer architectures demand a fresh look at the problems being solved and new algorithms to yield good performance. We propose a parallel sorting algorithm that moves a minimal amount of data over the network. Our algorithm is close to optimal in both the computation and the communication required. It moves less data than sample sorting algorithms, and is computationally efficient on distributed and shared memory architectures.

Blelloch et al. [20] compare several parallel sorting algorithms on the Connection Machine 2. They report that a sampling based sort and a radix sort are good algorithms to use in practice. We first tried a sampling based sort in which a number of sampled "splitter" elements are used to divide the input into buckets. The cost of sampling is often quite high, and sample sort requires a final redistribution phase so that the output has the desired distribution. The sampling process itself requires well chosen parameters to yield good samples. We noticed that we can do away with both these steps if exact splitters are found. Saukas and Song [105] describe a parallel selection algorithm. Our algorithm extends this work to efficiently find $p - 1$ exact splitters in $O(\log n)$ rounds of communication.

Our goal was to design a scalable, portable, and high performance sorting code that would form a building block for higher level combinatorial algorithms. We built our code using standards based library software such as the C++ Standard Template Library (STL) [98] and MPI [44]. Our code is highly modular, which lets the user replace any stage of the algorithm with platform or application specific routines for higher performance.

## 4.2 Algorithm Description

We have $p$ processors to sort $n$ total elements in a vector $v$. Assume that the input elements are already load balanced, or evenly distributed over the $p$ processors; this is not a requirement but makes the description and analysis simpler. We number the processors $1 \dots p$, and define $v_i$ to be the set of elements held locally by processor $i$. The *distribution* of $v$ is a vector $d$ where $d_i = |v_i|$. We say $v$ is *evenly distributed*if it is formed by the concatenation $v = v_1 \dots v_p$, and $d_i \leq \lceil \frac{n}{p} \rceil$ for all $i$.

```
Algorithm.
Input: A vector $v$ of $n$ total elements, evenly distributed
among $p$ processors.
Output: An evenly distributed vector $w$ with the same
distribution as $v$, containing the elements of $v$ in sorted
order.

  1. Locally sort the local elements $v_i$ into a vector $v_i'$.

  2. Determine the exact splitting of the local data:

      (a) Compute the partial sums and $r_j = \sum_{k=0}^{j} d_k$ for
          $j = 0 \ldots p$.

      (b) Use a parallel select algorithm to find the elements
          $e_1, \ldots, e_{p-1}$ of global rank $r_1, \ldots, r_{p-1}$, respectively.

      (c) For each $r_j$, have processor $i$ compute the local index
          $s_{ij}$ so that $r_j = \sum_{i=1}^{p} s_{ij}$ and the first $s_{ij}$ elements of
          $v_i'$ are no larger than $e_j$.

  3. Route the sorted elements in $v_i'$ according to the indices
     $s_{ij}$: processor $i$ sends elements in the range $s_{ij-1} \ldots s_{ij}$ to
     processor $j$.

  4. Locally merge the $p$ sorted sub-vectors into the output
     $w_i$.
```

**Figure 4.1:** Parallel sorting with exact splitters

We describe our algorithm in figure 4.1. We assume the task is to sort the input into increasing order. Of course, any other comparison function may be used.

### 4.2.1 Local sort

The first step may use any local sort applicable to the problem at hand. It is beyond the scope of this study to devise an efficient sequential sorting algorithm, as this problem is very well studied. Define the computation cost for the local sort on an input of size $n$ to be $T_s(n)$. Therefore, the amount of computation done by processor $i$ in this step is just $T_s(d_i)$. Since the local sorting must be completed on each processor before the next step can proceed, the global cost of the step is $\max_i T_s(d_i) = T_s(\lceil \frac{n}{p} \rceil)$. For a comparison based sort, this is $O(\frac{n}{p} \log \frac{n}{p})$.

### 4.2.2 Exact splitting

This step is nontrivial, and the main result of this chapter follows from the observation that exact global splitting over locally sorted data can be done efficiently.

The method we use for simultaneous selection is a generalization of the single selection by Saukas and Song [105], with two main differences: local ranking is done by binary search rather than partitioning, and we perform $O(\log n)$ rounds of communication rather than terminating the selection process earlier. For completeness, we describe the single selection algorithm next.

### Single selection

First, we consider the simpler problem of selecting just one target, an element of global rank $r$. The algorithm for this task is motivated by the sequential methods for the same problem, most notably the one given in [21].

Although it is simpler to define the selection algorithm recursively, the practical implementation and extension into simultaneous selection proceed more naturally from an iterative description. Define an *active range* to be the contiguous sequence of elements in $v_i'$ that may still have rank $r$, and let $a_i$ represent its size. Note that the total number of active elements is $\sum_{i=1}^{p} a_i$. Initially, the active range on each processor is its entire vector $v_i'$ and $a_i$ is just the input distribution $d_i$. In each iteration of the algorithm, a *pivot* is found that partitions the active range in two. Either the pivot is determined to be the target element, or the next iteration continues on one of the partitions. Every processor $i$ follows the steps in figure 4.2

We can think of the weighted median of medians as a pivot, because it is used to split the input for the next iteration. It is well known that the weighted median of medians can be computed in linear time [37, 102]. One possible way is to partition the values with the unweighted median, accumulate the weights on each side of the median, and recurse on the side that has too much weight.

**Algorithm.**
**Input:** Active range $v_i'$ on each processor.
**Output:** Weighted median of medians $m_m$.

```
1. Let $m_i$ be the median element of the active range of $v_i'$.
   Broadcast it to all processors.

2. Weigh median $m_i$ by $a_i / \sum_{k=1}^{p} a_k$. Find the weighted median
   of medians $m_m$. By definition, the weights of
   $\{m_i | m_i < m_m\}$ sum to at most $\frac{1}{2}$, as do the weights of
   $\{m_i | m_i > m_m\}$.

3. Use binary search over the active range of $v_i'$ to determine
   the first and last positions $f_i$ and $l_i$ that $m_m$ could be
   inserted into the sorted vector $v_i'$. Broadcast these two
   values.

4. Compute $f = \sum_{i=1}^{p} f_i$ and $l = \sum_{i=1}^{p} l_i$. The element $m_m$ has
   ranks $[f, l]$ in $v$.

5. If $r \in [f, l]$, then $m_m$ is the target element and we exit.
   Otherwise the active range is truncated with the
   following rule: Increase the bottom index to $l_i + 1$ if
   $l < r$; or decrease the top index to $f_i - 1$ if $r < f$. Repeat
   with truncated active range.
```

**Figure 4.2:** Parallel selection for the $k^{th}$ element

Therefore, the amount of computation in each round is $O(p)+O(\log a_i)+O(1) = O(p + \log \frac{n}{p})$ per processor.

Furthermore, splitting the data by the weighted median of medians will eliminate at least $\frac{1}{4}$ of the elements [105]. Since the step begins with $n$ elements under consideration, there are $O(\log n)$ iterations. The total single processor computation for selection is then $O(p \log n + \log \frac{n}{p} \log n) = O(p \log n + \log^2 n)$.

The amount of communication is straightforward to compute: two broadcasts per iteration, for $O(p \log n)$ total bytes being transferred in $O(\log n)$ rounds.

## Simultaneous selection

The problem is now to select multiple targets, each with a different global rank. In the context of the sorting problem, we want the $p-1$ elements of global rank $d_1, d_1 + d_2, \ldots, \sum_{i=1}^{p-1} d_i$. One simple way to do this would call the single selection problem for each desired rank. Unfortunately, doing so would increase the number of communication rounds by a factor of $O(p)$. We can avoid this inflation by solving multiple selection problems independently, but combining their communication. Stated another way, instead of finding $p-1$ paths one after another from root to leaf of the binary search tree, we take a breadth-first search with breadth at most $p-1$ (see figure 4.3).

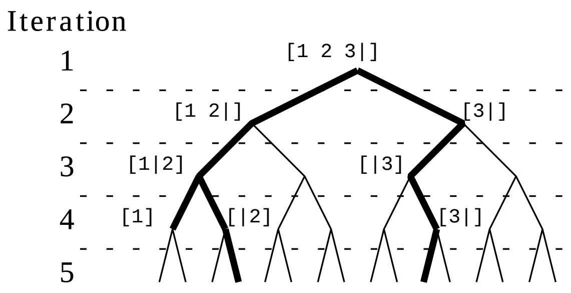


**Figure 4.3:** An example of selecting three elements. Each node corresponds to a contiguous range of $v_i'$, and gets split into its two children by the pivot. The root is the entire $v_i'$, and the bold edges trace which ranges are active at each iteration. The array at a node represents the target ranks that may be found by the search path, and the vertical bar in the array indicates the relative position of the pivot's rank.

To implement simultaneous selection, we augment the single selection algorithm with a set $A$ of active ranges. Each of these active ranges will produce at least one target. An iteration of the algorithm proceeds as in single selection, but finds multiple pivots: a weighted median of medians for each active range. If an active range produces a pivot that is one of the target elements, we eliminate that active range from $A$ (as in the leftmost branch of figure 4.3). Otherwise, we examine each of the two partitions induced by the pivot, and add it to $A$ if it may yield a target. Note that, as in iterations 1 and 3 in figure 4.3, it is possible for both partitions to be added.

In slightly more detail, we handle the augmentation by looping over $A$ in each step. The local medians are bundled together for a single broadcast at the end of Step 1, as are the local ranks in Step 3. For Step 5, we use the fact that each active range in $A$ has a corresponding set of the target ranks: those targets that lie between the bottom and top indices of the active range. If we keep the subset of target ranks sorted, a binary search over it with the pivot rank[1] will split the target set as well. The left target subset is associated with the left partition of the active range, and the right sides follow similarly. The left or right partition of the active range is added to $A$ for the next iteration only if the corresponding target subset is non-empty.

The computation volume for simultaneous selection follows by inflating each step of the single selection by a factor of $p$ (because $|A| \leq p$). The exception is the last step, where we also need to search over $O(p)$ targets. This amounts to $O(p + p^2 + p \log \frac{n}{p} + p + p \log p) = O(p^2 + p \log \frac{n}{p})$ per iteration. Again, there are $O(\log n)$ iterations, for total computation time of $O(p^2 \log n + p \log^2 n)$.

This step runs in $O(p)$ space, the scratch area needed to hold received data and pass state between iterations.

The communication time is similarly inflated: two broadcasts per round, each having one processor send $O(p)$ data to all the others. The aggregate amount of data being sent is $O(p^2 \log n)$ over $O(\log n)$ rounds.

[1]Actually, we search for the first position $f$ may be inserted, and for the last position $l$ may be inserted. If the two positions are not the same, we have found at least one target.

**Producing indices**

Each processor computes a local matrix $S = S_{ij}$ of size $p \times (p+1)$. Recall that $S$ splits the local data $v_i'$ into $p$ segments, with $s_{k0} = 0$ and $s_{kp} = d_k$ for $k = 1 \ldots p$. The remaining $p-1$ columns are the result of the selection. For simplicity of notation, we briefly describe the output procedure in the context of single selection; it extends naturally for simultaneous selection. When we find that a particular $m_m$ has global ranks $[f, l) \ni r_k$, we also have the local ranks $f_i$ and $l_i$. There are $r_k - f$ excess elements with value $m_m$ that should be routed to processor $k$. We assign $s_{ki}$ from $i = 1$ to $p$, taking as many elements as possible without overstepping the excess, which guarantees that our sort is stable (i.e. it preserves the ordering of equal elements). More precisely,

$$s_{ki} = \min \left\{ f_i + (r_k - f) - \sum_{j=1}^{i-1} (s_{kj} - f_j),\ l_i \right\}$$

.

The computation requirement for this step is $O(p^2)$ to populate the matrix $S$; the space used is also $O(p^2)$.

### 4.2.3 Element routing

Any parallel sorting algorithm must move elements from the locations they start out to where they belong in sorted order. An optimal parallel sorting

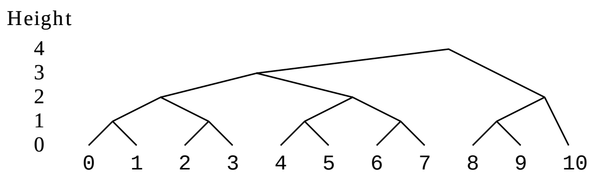


**Figure 4.4:** An example of tree merging when the number of processors is not a power of 2.

algorithm will communicate every element from its current location to a location in remote memory at most once. Our algorithm is optimal in this sense, not counting the $O(p^2 \log n)$ data movement during selection, which is of much lower order than $n$ for realistic values of $n$ and $p$. For instance, if the input is already sorted, no data movement occurs. However, if the input is in reverse sorted order, all elements need to be communicated to their destinations. Then the amount of data communicated in the element routing step is $\Theta(n)$.

### 4.2.4 Merging

Now each processor has $p$ sorted sub-vectors, and we want to merge them into a single sorted sequence. Conceptually, we build a binary tree on top of the vectors. To handle the case where $p$ is not a power of 2, we require that a node of height $i$ has at most $2^i$ leaf descendants, whose ranks are in $[k \cdot 2^i, (k+1) \cdot 2^i)$ for some $k$ (figure 4.4). It is clear that the tree has height at most $\lceil \log p \rceil$.

For reasons of cache efficiency, we merge pairs of sub-vectors out-of-place from this tree. Cache oblivious algorithms [24, 52] may yield better performance across a variety of architectures.

Notice that a merge will move a particular element exactly once (from one buffer to its sorted position in the other buffer). Furthermore, there is at most one comparison for each element move. Finally, every time an element is moved, it goes into a sorted sub-vector at a higher level in the tree. Therefore each element moves at most $\lceil \log p \rceil$ times, for a total computation time of $d_i \lceil \log p \rceil$. Again, we take the time of the slowest processor, for overall parallel computation time of $\lceil \frac{n}{p} \rceil \lceil \log p \rceil$.

## 4.3 Theoretical performance

Suppose the sequential sorting algorithm used locally on each processor takes time $T_s(n)$. We define $T_s^*(n, p) = \frac{1}{p} T_s(n)$, which would be the ideal running time of our parallel algorithm with perfect speedup. Adding up the time complexity of each step of our parallel algorithm gives its total computation time:

$$T_s^*(n, p) + O(p^2 \log n + p \log^2 n) + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of 2}) \tag{4.1}$$

The total space used in addition to the input, is $O(p^2 + \frac{n}{p})$.

We compare this algorithm against an ideal parallel sorting algorithm with the following properties:

1. Total computation time $T^*_s(n,p) = \frac{1}{p}T_s(n)$. Linear speedup in $p$ over a sequential sorting algorithm with running time $T_s(n)$.

2. Minimum amount of cross-processor communication $T^*_c(v)$, the number of elements that begin and end on different processors.

If $T_s(n)$ is optimal, then so is $T^*_s(n,p)$. If there were a faster $T'_s(n,p)$ for some $p$, then we could simulate it on a single processor for total time $pT'_s(n,p) < pT^*_s(n,p) = T_s(n)$, which is a contradiction.

## 4.3.1 Analysis of computation time

We can determine the total computation time by adding up the time for each step, and comparing against the theoretical $T^*_s(n,p)$:

$$\begin{aligned} & T_s(\lceil \tfrac{n}{p} \rceil) + O(p^2 \log n + p \log^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \log p \rceil \\ \leq \; & \frac{1}{p} T_s(n+p) + O(p^2 \log n + p \log^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \log p \rceil \\ = \; & T^*_s(n+p,p) + O(p^2 \log n + p \log^2 n) + \lceil \tfrac{n}{p} \rceil \lceil \log p \rceil \end{aligned}$$

The inequality follows from the fact that $T^*_s(n) = \Omega(n)$.

It is interesting to consider the case where the local sort is comparison based. Then the sequential sort has $T_s(n) \leq c\lceil \frac{n}{p} \rceil \log\lceil \frac{n}{p} \rceil$ for some $c \geq 1$. We can then add this cost to the time required for merging (Step 4):

$$\begin{aligned} & c\lceil \tfrac{n}{p} \rceil \log \lceil \tfrac{n}{p} \rceil + \lceil \tfrac{n}{p} \rceil \lceil \log p \rceil \\ \leq\ & c\lceil \tfrac{n}{p} \rceil \log(n+p) + \lceil \tfrac{n}{p} \rceil (\lceil \log p \rceil - c \log p) \\ \leq\ & c\lceil \tfrac{n}{p} \rceil \log n + c\lceil \tfrac{n}{p} \rceil \log(1 + \tfrac{p}{n}) + \lceil \tfrac{n}{p} \rceil (\lceil \log p \rceil - c \log p) \\ \leq\ & \frac{cn \log n}{p} + \log n + 2c + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of 2}) \end{aligned}$$

With comparison sorting, the total computation time becomes:

$$T_s^*(n,p) + O(p^2 \log n + p \log^2 n) + (\lceil \tfrac{n}{p} \rceil \text{ if } p \text{ not a power of 2}) \tag{4.2}$$

### 4.3.2 Analysis of communication volume

We have already established that the exact splitting algorithm will provide the final locations of the elements. The amount of communication done in the routing phase is then the optimal amount. Therefore, total communication volume is:

$$T_c^*(v) \text{ in 1 round} + O(p^2 \log n) \text{ in } \log n \text{ rounds}$$

### 4.3.3 Realistic assumptions

These bounds imply that our algorithm is only efficient for $p^2 \leq n/p$ which implies that $p^3 \leq n$. This requirement is common to other parallel sorting algorithms, such as sampling based sorting algorithms [62]. It is a realistic assumption for the case when the amount of data to be sorted is a significant fraction of the total memory on all processors, and a single processor has more than $p^2$ memory.

## 4.4 Experimental results

The communication cost of our sorting algorithm is nearly optimal if $p$ is small. Furthermore, the computation speedup is nearly linear if $p \ll n$. Notice that the speedup is given with respect to a sequential algorithm, rather than to itself with small $p$. Our intent is that efficient sequential sorting algorithms and implementations can be developed without any consideration for parallelization, and then be substituted in our implementation for good parallel performance.

We now turn to empirical results, which suggest that exact splitting uses little computation and communication time.

### 4.4.1 Experimental setup

We implemented our parallel sorting algorithm in C++ using the Message Passing Interface (MPI) [44] for communication. We intend our code to be used as a library with a simple interface; it is therefore templated, and comparison based.

We use `std::sort` and `std::stable_sort` from the C++ Standard Template Library (STL) library for sequential sorting.The C++ STL has one of the fastest general purpose sorting routines available [98].

We use MPI for communication. MPI is the most portable and widely used method for communication in parallel computing. Since vendor optimized MPI implementations are available on most platforms, we expect reasonable performance on distributed as well as shared memory architectures. We use the MPI libraries provided by the SGI MPT [106] on the Altix, and OpenMPI [54] on clusters.

Our choice of the C++ STL sequential sorting routines and MPI allows our code to be robust, scalable and portable without sacrificing performance. We tested our implementation on an SGI Altix and a commodity cluster. The SGI Altix had 256 Itanium 2 processors and 4TB of RAM in a single system image. The Beowulf cluster had 32 Xeon processors and 3 GB of memory per node connected via gigabit ethernet.

We ran every test instance twice, timing only the second invocation. As a result, setup time such as page table initializations etc. are not counted in our timings.

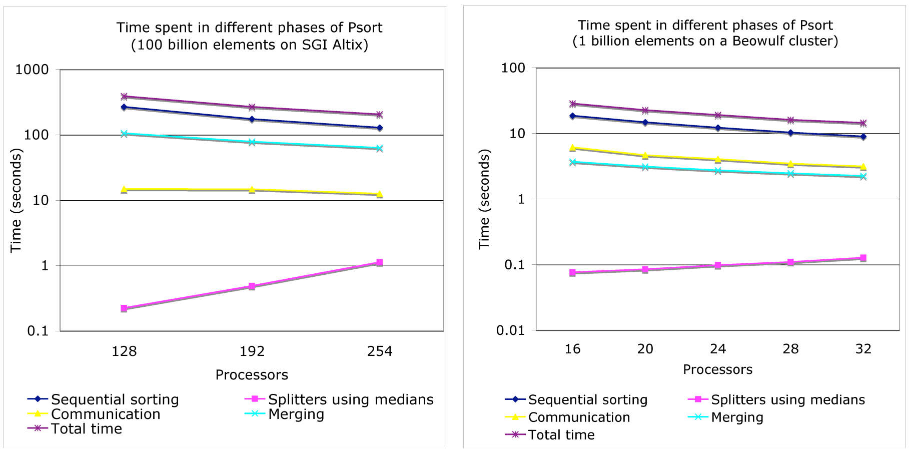


**Figure 4.5:** Scalability tests are performed for a fixed problem size, while changing the number of processors. Good scaling is observed on shared memory architectures (left) as well as on clusters (right).

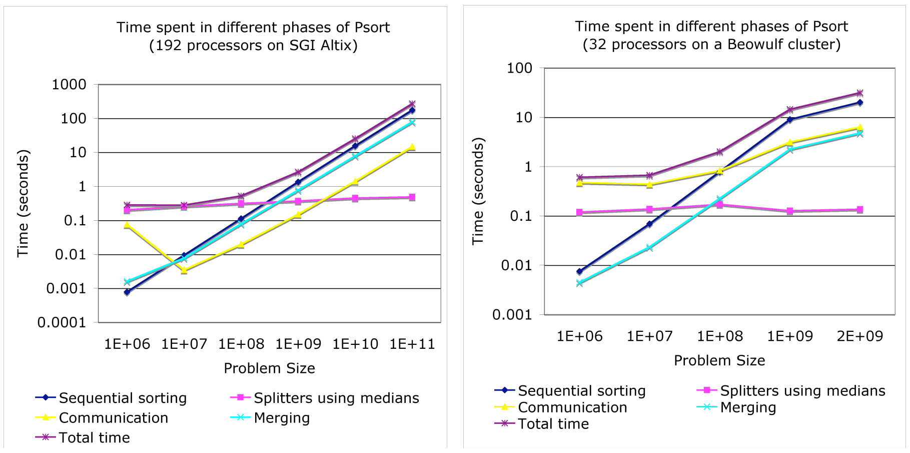


**Figure 4.6:** Scalability tests are performed for a fixed problem size while changing the number of processors. Good scaling is observed on large problems on shared memory architectures (left) as well as on clusters (right).

Figure 4.5 shows the scaling of our algorithm as the same number of elements are sorted on different numbers of processors. We do not provide comparisons to sequential performance because the datasets do not fit on a single processor. The largest problem we solved is to sort 100 billion elements with 254 processors, which took less than 4 minutes.

Figure 4.6 shows the scaling of our algorithm with the problem size, the number of processors being fixed. In all the cases, we observe good scaling on large problem sizes on a shared memory Altix as well as on a cluster.

For clusters, we also present sequential speedup in figure 4.7. We do not observe good scaling on small problems. As the problems get larger, we observe better scaling, as expected. For the largest problem size (1 billion), it is not possible to run the code on small numbers of processors. We extrapolate the performance for small numbers of processors, and present actual performance for 16 processors and higher.

We also experimented with cache oblivious strategies. We tried using funnel sort [25] for sequential sorting, but found it to be slower than the STL sorting algorithms. On the other hand, a cache oblivious funnel merge did yield slightly better performance than the out-of-place tree merge we use in our code. We compare the performance of the two merging algorithms on the Altix in figure 4.8.

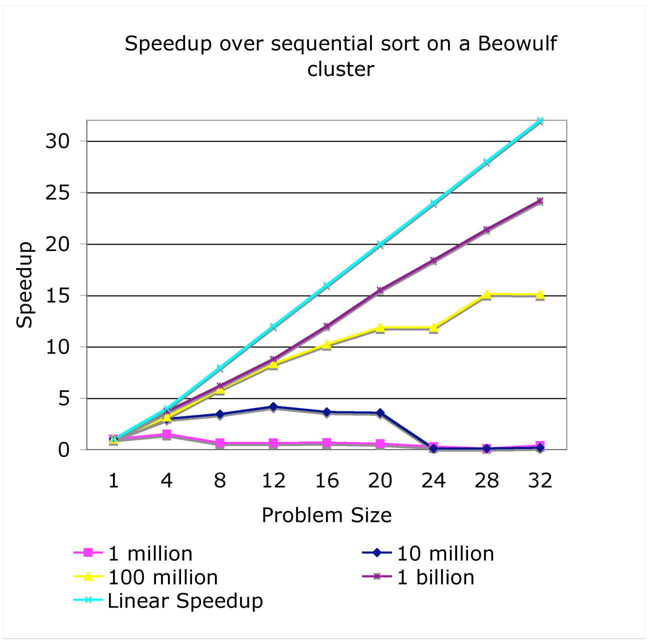


**Figure 4.7:** The 45 degree line represents perfect scaling. As the problem size gets larger, better scaling is observed. However, for small to moderate sized problems, the scaling is poor, as expected on beowulf clusters.

## 4.4.2 Comparison with sample sorting

Several prior works [20, 68, 109] conclude that sample sort is the most efficient parallel sorting algorithm for large $n$ and $p$. Samplesort is characterized by having each processor distribute its $\lceil \frac{n}{p} \rceil$ elements into $p$ buckets, where the bucket boundaries are determined by some form of sampling. Once the buckets are formed, a single round of all-to-all communication follows, with each processor $i$ receiving the contents of the $i$th bucket from everybody else. Finally, each processor performs some local computation to place all its received elements in sorted order.

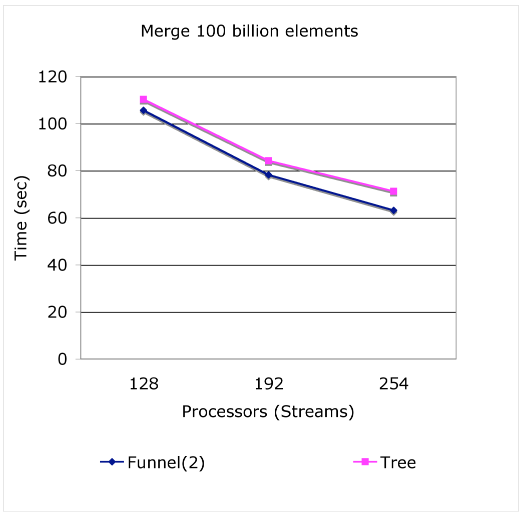


**Figure 4.8:** The performance of cache-oblivious merging is compared with simple tree based merging. Cache oblivious outperforms tree merging on very large problem sizes with a large number of processors by up to 15%. The savings are much smaller as a fraction of total sorting time.

One of the problems we encountered with sample sorting was the cost of picking samples, and picking splitters from those samples. Since we are interested in sorting extremely large amounts of data, the sampling step and picking splitters turns out to be very expensive.

The other drawback of sample sort is that the final distribution of elements may be uneven. Much of the work in sample sorting is directed towards reducing the amount of imbalance, providing schemes that have theoretical bounds on the largest amount of data a processor can collect in the routing. The problem with one processor receiving too much data is that the computation time in the subsequent steps is dominated by this one overloaded processor. Furthermore,

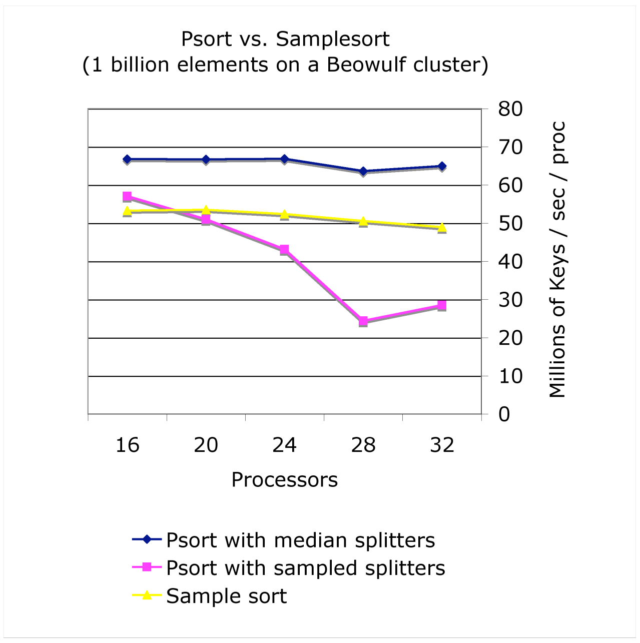


**Figure 4.9:** Several parallel sorting algorithms are compared. A parallel sort with exact splitters determined through parallel selection outperforms both implementations of sample sort.

some applications require an exact output distribution; this is often the case when sorting is just one part of a multi-step process. In such cases, an additional redistribution step would be necessary, where elements are communicated across boundaries.

We compare the performance of our algorithm with two different implementations of sampling based sorts in Figure 4.9. "Psort with median splitters" is our parallel sorting algorithm, which uses medians on each processor to pick exact splitters. "Psort with sampled splitters" is the same algorithm, but it uses random sampling to pick splitters instead of medians. “Sample sort” is a traditional sampling based sorting algorithm, which has the following steps:

1. Pick splitters by sampling or oversampling.
2. Partition local data to prepare for the communication phase.
3. Route elements to their destinations.
4. Sort local data.
5. Redistribute to adjust processor boundaries.

The steps in Samplesort differ from the Psort algorithms in two ways. Psort sorts local data first, whereas Sample sort sorts local data as the last step in the algorithm. Sample sort may need to do an extra round of communication to adjust processor boundaries if the resulting distribution is different from the required one.

Our algorithm runs much faster than the traditional Samplesort, although both show good scalability. We do not experiment with a wide range of sampling methods, but follow a standard strategy of picking $p^2$ splitters for $p$ processors. Since our input is uniformly distributed, we do get good splitters.

1. Partitioning local data in Sample sort before the element routing step is much slower than merging streams of data received from other processors in Psort. This is because merging is much more cache-friendly than partitioning.

2. Sample sort also has to perform an extra round of communication to re-balance the data distribution. If the sampled splitters do not approximate the distribution well, the load imbalance may be large, and this extra round of communication may incur a larger penalty.

## 4.5 Conclusion

We present a high performance, highly scalable parallel sorting algorithm that compares favorably against the traditional sample sort algorithm. Our code uses the C++ Standard Template Library [98] and MPI [44], making it robust and portable.

There may be room for further improvement in our implementation. The cost of merging can be reduced by interleaving the $p$-way merge step with the element routing, merging sub-arrays as they are received. Alternatively, using a data structure such as a funnel [24, 52] may allow better cache efficiency to reduce the merging time. Another potential area of improvement is the exact splitting. Instead of traversing the search tree to completion, a threshold can be set; when the active range becomes small enough, a single processor gathers all the remaining active elements and completes the splitter computation sequentially. This method, used by Saukas and Song [105], helps reduce the number of communication rounds in the tail end of the step. Finally, this parallel sorting algorithm will directly benefit from future improvements in sequential sorting and communication schemes.

This is a new deterministic algorithm for parallel sorting that makes a strong case for exact splitting on modern high performance computers. Aside from some intricacies of determining the exact splitters, the algorithm is conceptually simple to understand, analyze, and implement. Our implementation is used to provide the `sort` function in STAR-P [76], and we hope our efforts will guide other implementations.

# Chapter 5

# Applications of Star-P and the Graph Toolbox

This chapter describes several applications built with STAR-P and the graph toolbox. The following applications are described: a problem in computational fluid dynamics, a graph analysis benchmark implemented on extremely large graphs, combinatorial preconditioners for solving linear systems, and non-negative matrix factorizations.

## 5.1 An application in computational fluid dynamics

This application used a pre-release version of STAR-P when `eigs` was not yet available to solve eigenvalue problems.

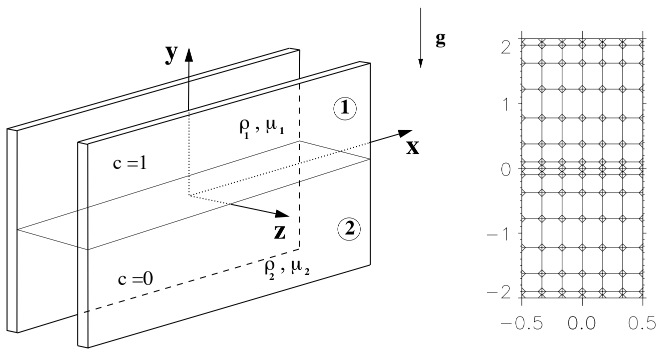


**Figure 5.1:** Geometry of the Hele-Shaw cell. The heavier fluid is placed above the lighter one. Either one of the fluids can be the more viscous one. A Chebyshev grid is employed in the $y$-direction, and compact finite differences in the $z$-direction.

Goyal and Meiburg [64] study the influence of viscosity variations on the density driven instability of two miscible fluids. The two fluids of different density and viscosity are in a vertical Hele-Shaw cell as shown in figure 5.1. This problem is used to model porous media flows and finds applications in enhanced oil recovery, fixed bed regeneration and groundwater flows.

Figure 5.1 shows the discretization of the problem, which yields an algebraic system of the form $A\phi = \sigma B\phi$. The eigenvalue $\sigma$ represents the growth rate of the perturbations, while the eigenvector $\phi$ reflects the shape of the perturbations. A positive (negative) eigenvalue indicates unstable (stable) behavior. The system has a $5 \times 5$ block structure reflecting the 5 variables at each mesh point (3 velocity components $u$, $v$ and $w$, relative concentration of the heavier fluid $c$, and pressure $p$).

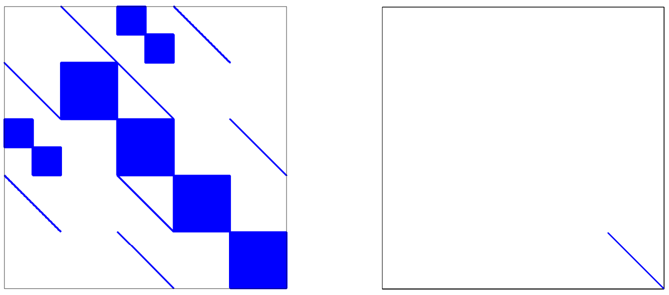


**Figure 5.2:** The generalized eigenvalue problem to be solved for the Hele-Shaw cell is $A\phi = \sigma B\phi$. $A$ is an asymmetric matrix, whereas $B$ is a diagonal matrix with many zeros on the diagonal

A discretization of $165 \times 25$ points turns out to be sufficient for this problem. Since we solve for 5 variables at each grid point, the matrix $A$ is of the size $20,625 \times 20,625$. The number of non-zeros is $3,812,450$. The matrix is unsymmetric both in values and non-zero structure, as shown in the spy plots in figure 5.2. In order to calculate the largest eigenvalue, we use the power method with shift and invert in STAR-P.

The original non-STAR-P code used LAPACK [7] with ARPACK [84], while the STAR-P code is using a shift-and-invert method with a linear solve (SuperLU_DIST) as shown in figure 5.3. Note that this problem was solved before `eigs` was supported in STAR-P. Hence we used a simple STAR-P implementation of the power method. We use an initial guess of 0.1, and it converges to 0.0194. The precision is sufficient for linear stability analysis. We used a cluster[1]

[1]We used 16 nodes of a 32-node cluster. Each node in the cluster had a 2.6GHz Pentium Xeon CPU, 3GB of RAM, and gigabit ethernet.

```
function lambda = power (A, B, sigma, iter, tol)
% Power method to compute the largest eigenvalue

n = length (A);
C = A - sigma * B;
y = rand (n, 1);

for k=1:iter
  q = y ./ norm (y);
  v = B * q;
  y = C \ v;

  theta = dot (q, y);
  res = norm (y - theta * q);
  if res <= tol,  break;   end
end

lambda = 1 / theta + sigma;
```

**Figure 5.3:** Star-P code for the shift-and-invert eigensolver.

for the computation. Results are presented in Table 5.1. Our results validated the results attained from the original sequential code, and allow scaling to larger problems.

## 5.2 SSCA #2 graph analysis benchmark

The SSCAs (Scalable Synthetic Compact Applications) are a set of benchmarks designed to complement existing benchmarks such as the HPL [43] and the NAS parallel benchmarks [13]. Specifically, SSCA #2 [10] is a compact ap-

| No. of processors | Time (seconds) |
|---|---|
| 4 | 90 |
| 8 | 39 |
| 16 | 33 |

**Table 5.1:** Time to solve the generalized eigenvalue problem. With a large number of processors, excess communication within the Beowulf cluster in the sparse direct solver reduces performance.

plication that has multiple kernels accessing a single data structure (a directed multigraph with weighted edges). We describe our implementation of version 1.1 of the benchmark. Version 2.0 [12], which differs significantly from version 1.1, has since been released.

The data generator generates an edge list in random order for a multigraph of sparsely connected cliques as shown in Figure 5.4. The four kernels are as follows:

1. Kernel 1: Create a data structure for further kernels.
2. Kernel 2: Search graph for a maximum weight edge.
3. Kernel 3: Perform breadth first searches from a set of start nodes.
4. Kernel 4: Recover the underlying clique structure from the undirected graph.

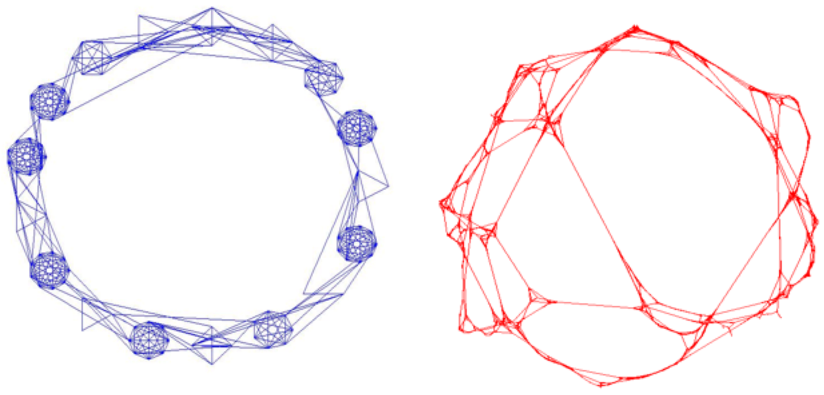


**Figure 5.4:** The left image shows the conceptual SSCA #2 graph (Kepner). The image on the right is an SSCA #2 graph generated with scale 8 (256 nodes) and plotted with Fiedler co-ordinates.

## 5.2.1 SSCA#2 description

### Scalable data generator

The data generator is the most complex part of our implementation. It generates edge tuples for subsequent kernels. No graph operations are performed at this stage. The input to the data generator is a `scale` parameter, which indicates the size of the graph being generated. The resulting graph has $2^{scale}$ nodes, with a maximum clique size of $\lfloor 2^{scale/3} \rfloor$, a maximum of 3 edges with the same endpoints, and a probability of 0.2 that an edge is uni-directional. The vertex numbers are randomized, and a randomized ordering of the edge tuples is presented to the subsequent kernels. Our implementation of the data generator closely follows the pseudocode published in the spec [10].

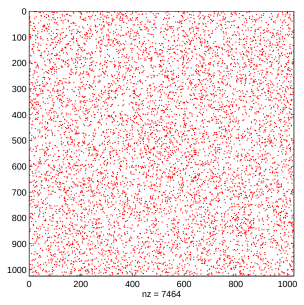


**Figure 5.5:** MATLAB spy plot of the input graph. The input graph is randomized, as evidenced by no observed patterns in the spy plot.

### Kernel 1

Kernel 1 creates a read-only data structure that is used by subsequent kernels. We create a sparse matrix corresponding to each layer of the multigraph. The multigraph has three layers, since there is a maximum of three parallel edges between any two nodes in the graph. MATLAB provides several ways of constructing sparse matrices, `sparse`,which takes as its input a list of three-tuples: $(i, j, w_{ij})$. Its output is a sparse matrix with a nonzero $w_{ij}$ in every location $(i, j)$ specified in the input. Figure 5.5 shows a spy plot of one layer of the input graph.

### Kernel 2

In kernel 2, we search the graph for edges with maximum weight. `find` is the inverse of `sparse`. It returns all nonzeros from a sparse matrix as a list of three-tuples. We then use `max` to find the maximum weight edge.

### Kernel 3

In kernel 3, we perform breadth first searches from a given set of starting points. We use sparse matrix matrix multiplication to perform all breadth first searches simultaneously from the given starting points. Let $G$ be the adjacency matrix representing the graph and $S$ be a matrix corresponding to the starting points. $S$ has one column for each starting point, and one non-zero in each column. Breadth first search is performed by repeatedly multiplying $G$ by $S$: $Y = G * S$. We perform several breadth first searches simultaneously by using sparse matrix matrix multiplication. Star-P stores sparse matrices by rows, and parallelism is achieved by each processor computing some rows in the product [103, 108].

### Kernel 4

Kernel 4 is the most interesting part of the benchmark. It can be considered to be a partitioning problem or a clustering problem. We have several

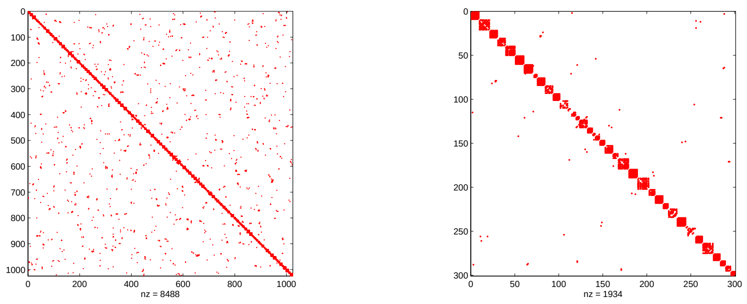


**Figure 5.6:** The image on the left is a spy plot of the graph, reordered after clustering. The image on the right magnifies a portion around the diagonal. Cliques are revealed as dense blocks on the diagonal.

implementations of kernel 4 based on spectral partitioning (figure 5.4), "seed growing" (figure 5.6), and "peer pressure" algorithms. The peer pressure and seed growing implementations scale better than the spectral methods, as expected. We now demonstrate how we use the infrastructure described above to implement kernel 4 in a few lines of Matlab or Star-P. Figure 5.6 shows a spy plot of the undirected graph after clustering. The clusters show up as dense blocks along the diagonal.

Our seed growing algorithm (figure 5.7) starts by picking a small set of seeds (about 2% of the total number of nodes) randomly. The seeds are then grown so that each seed claims all nodes reachable by at least $k$ paths of length 1 or 2, where $k$ is the size of the largest clique. This may cause some ambiguity, since some nodes might be claimed by multiple seeds. We tried picking an independent set of nodes from the graph by performing one round of Luby's al-

```
function J = seedgrow (seeds)
% Clustering by breadth first searches

% J is a sparse matrix with one seed per column.
J = sparse (seeds, 1:nseeds, 1, n, nseeds);

% Vertices reachable with 1 hop.
J = G * J;
% Vertices reachable with 1 or 2 hops.
J = J + G*J;
% Vertices reachable with at least k paths of 1 or 2 hops.
J = J >= k;
```

**Figure 5.7:** Breadth first parallel clustering by seed growing.

gorithm [87] to keep the number of such ambiguities as low as possible. However, the quality of clustering remains unchanged when we use random sampling. We use a simple approach for disambiguation: the lowest numbered cluster claiming a vertex claims it. We also experimented with attaching singleton nodes to nearby clusters to improve the quality of clustering.

Our peer pressure algorithm (figure 5.8) starts with a subset of nodes designated as leaders. There has to be at least one leader neighboring every vertex in the graph. This is followed by a round of voting where every vertex in the graph selects a leader, selecting a cluster to join. This does not yet yield good clustering. Each vertex now looks at its neighbors and switches its vote to the most popular leader in its neighborhood. This last step is crucial, and in this case, it recovers more than 95% of the original clique structure of the graph.

```
function cluster = peerpressure (G)
% Clustering by peer pressure

% Use maximal independent set routine from GAPDT
IS = mis (G);

% Find all neighbors in the independent set.
neighbors = G * sparse(IS, IS, 1, length(G), length(G));

% Each vertex chooses a random neighbor in the independent set.
R = sprand (neighbors);
[ignore, vote] = max (R, [], 2);

% Collect neighbor votes and join the most popular cluster.
[I, J] = find (G);
S = sparse (I, vote(J), 1, n, n);
[ignore, cluster] = max (S, [], 2);
```

**Figure 5.8:** Parallel clustering by peer pressure

We experimented with different approaches to select leaders. At first, it seemed that a maximal independent set of nodes from the graph was a natural way to pick leaders. In practice, it turns out that simple heuristics (such as the highest numbered neighbor) gave equally good clustering. We also experimented with more than one round of voting. The marginal improvement in the quality of clustering was not worth the additional computation time.

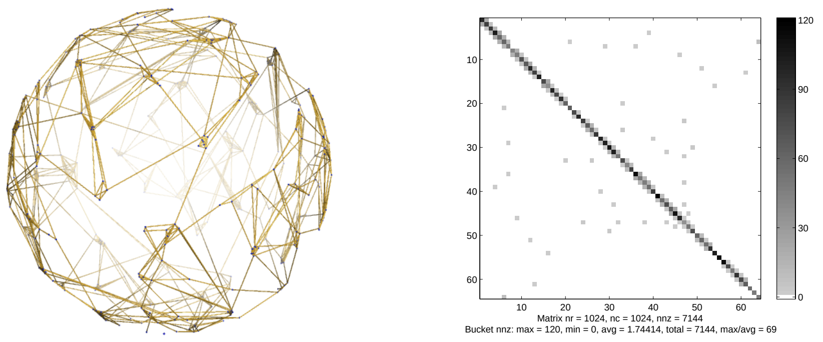


**Figure 5.9:** The 3D visualization of the SSCA #2 graph on the left is produced by relaxing the Fiedler co-ordinates projected onto the surface of a sphere. The right figure shows a density spy plot of the SSCA #2 graph.

### 5.2.2 Visualization of large graphs

Graphics and visualization are a key part of an interactive system such as Matlab. The question of how to effectively visualize large datasets in general, especially large graphs, is still unsolved. We successfully applied methods from numerical computing to come up with meaningful visualizations of the SSCA #2 graph.

One way to compute geometric co-ordinates for the nodes of a connected graph is to use Fiedler co-ordinates [67] for the graph. Figure 5.4 shows the Fiedler embedding of the SSCA #2 graph. In the 2D case, we use the eigenvectors (Fiedler vectors) corresponding to the first two non-zero eigenvalues of the Laplacian matrix of the graph as co-ordinates for nodes of the graph in a plane.

For 3D visualization of the SSCA #2 graph, we start with 3D Fiedler co-ordinates projected onto the surface of a sphere. We model nodes of the graph as particles on the surface of a sphere. There is a repulsive force between all particles, inversely proportional to the distance between them. Since these particles repel each other on the surface of a sphere, we expect them to spread around and occupy the entire surface of the sphere. Since there are cliques in the original graph, we expect clusters of particles to form on the surface of the sphere. At each timestep, we compute a force vector between all pairs of particles. Each particle is then displaced some distance based on its force vector. All displaced particles are projected back onto the sphere at the end of each timestep.

This algorithm was used to generate figure 5.9. In this case, we simulated 256 particles and the system was evolved for 20 timesteps. It is important to first calculate the Fiedler co-ordinates. Beginning with random co-ordinates results in a meaningless picture. We used PyMOL [40] to render the graph.

### 5.2.3 Experimental Results

We ran our implementation of SSCA #2 (ver 1.1, integer only) in STAR-P. The MATLAB client was run on a generic PC. The STAR-P server was run on an SGI Altix with 128 Itanium II processors with 128G RAM (total, non-uniform

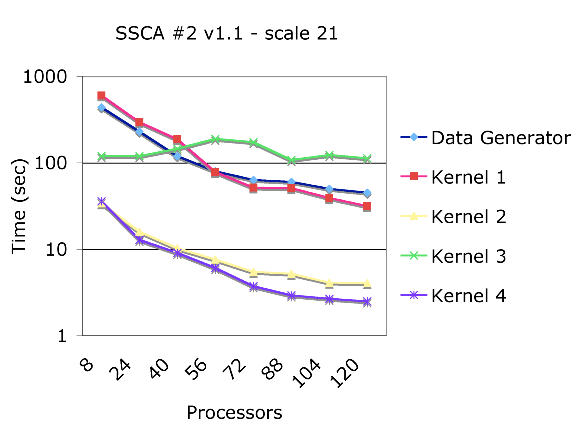


**Figure 5.10:** SSCA #2 v1.1 execution times (STAR-P, Scale=21)

memory access). We used a graph generated with scale 21. This graph has 2 million nodes. The multigraph has 321 million directed edges; the undirected graph corresponding to the multigraph has 89 million edges. There are 32 thousand cliques in the graph, the largest having 128 nodes. There are 89 million undirected edges within cliques, and 212 thousand undirected edges between cliques in the input graph for kernel 4. The results are presented in Fig. 5.10.

Our data generator scales well; the benchmark specification does not require the data generator to be timed. A lot of time is spent in kernel 1, where data structures for the subsequent kernels are created. The majority of this time is spent in searching the input triples for duplicates, since the input graph is a multigraph. Kernel 1 creates several sparse matrices using `sparse`, each corresponding to a layer in the multigraph. Time spent in kernel 1 also scales

| Operation | Source LOC | Total line counts |
|---|---|---|
| Data generator | 176 | 348 |
| Kernel 1 | 25 | 63 |
| Kernel 2 | 11 | 34 |
| Kernel 3 | 23 | 48 |
| Kernel 4 (spectral) | 22 | 70 |
| Kernel 4 (seed growing) | 55 | 108 |
| Kernel 4 (peer pressure) | 6 | 29 |

**Table 5.2:** Line counts for Star-P implementation of SSCA#2. The "Source LOC" column counts only executable lines of code, while the "Total line counts" column counts the total number of lines including comments and whitespace.

very well with the number of processors. Time spent in Kernel 2 also scales as expected.

Kernel 3 does not show speedups at all. Although all the breadth first searches are performed in parallel, the process of subgraph extraction for each starting point creates a lot of traffic between the Star-P client and the Star-P server, which are physically in different states. This client server communication time ends up dominating over the computation time. We will minimize this overhead by vectorizing all of kernel 3 in a future release.

Kernel 4, the non-trivial part of the benchmark, actually scales very well. We show results for our best performing implementation of kernel 4, which uses the seed growing algorithm.

The evaluation criteria for the SSCAs also include software engineering metrics such as code size, readability, maintainability, etc. Our implementation is extremely concise. We show the source lines of code (SLOC) for our implementation in Table 5.2. We also show absolute line counts, which include blank lines and comments, as we believe these to be crucial for code readability and maintainability. Our implementation runs without modification in sequential MATLAB, making it easy to develop and debug on the desktop before deploying on a parallel platform.

We have run the full SSCA #2 benchmark (spec v0.9, integer only) on graphs with $2^{27} = 134$ million nodes on the SGI Altix. We have also manipulated extremely large graphs (1 billion nodes and 8 billion edges) on an SGI Altix with 256 processors using STAR-P.

This demonstrates that the sparse matrix representation is a scalable and efficient way to manipulate large graphs. Not that the codes in figure 5.7 and figure 5.8 are not pseudocodes, but actual code excerpts from our implementation. Although the code fragments look very simple and structured, the computation manipulates sparse matrices, resulting in highly irregular communication patterns on irregular data structures.

## 5.3 Solution of sparse linear systems

We describe the implementation of two types of combinatorial preconditioners: support graph (Vaidya) preconditioners and an algebraic multigrid preconditioner.

Sparse linear systems arise naturally in several problems, most commonly in the numerical solution of partial differential equations. Preconditioned iterative methods are often the methods of choice to solve large sparse linear systems. The number of iterations of conjugate gradient (CG) needed to solve $Ax = b$ is bounded above by the square root of the condition number of $A$ [41]. Convergence can be accelerated by solving a preconditioned linear system $M^{-1}Ax = M^{-1}b$ instead [120]. The number of iterations of CG is then bounded by the square root of the ratio of the extreme generalized eigenvalues of $Ay = \lambda My$. Combinatorial preconditioners appear to provide a promising approach to solve certain classes of problems [17, 22, 30, 119].

### 5.3.1 Support graph preconditioners

Vaidya proposed two classes of preconditioners [18]. The first class, maximum weight spanning tree (MWST) preconditioners, guarantees a condition number bound of $O(n^2)$ for any $n \times n$ sparse diagonally dominant symmetric

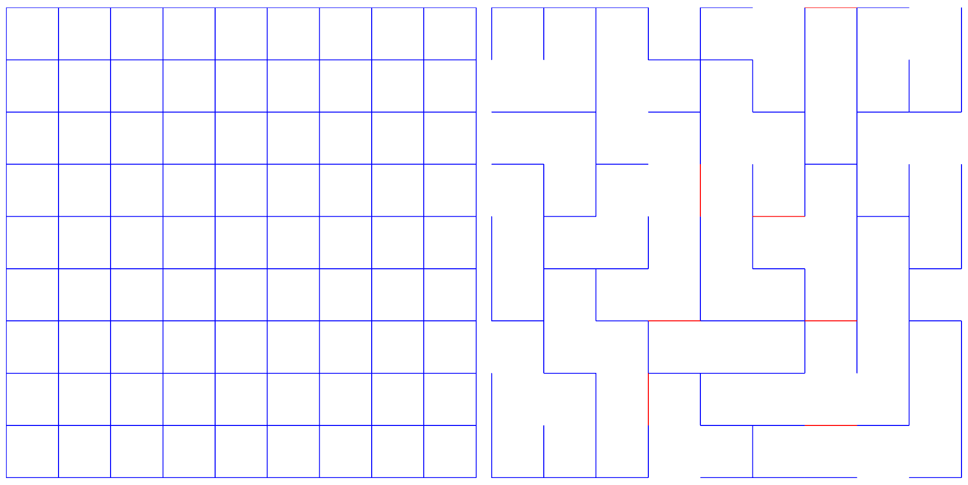


**Figure 5.11:** The 2D model problem and the corresponding Vaidya preconditioner. The basic Vaidya preconditioner is shown in blue; the augmented Vaidya precondtioner includes the extra red edges.

$M$-matrix. The second class augments the MWST with extra edges. These extra edges can be found quickly (sequentially) with simple graph algorithms. The cost of factorizing these preconditioners depends upon the extra edges added. The factorization cost can be balanced with iteration costs, bounding the work in the linear solver by $(O^{1.75})$ for arbitrary sparse $M$-matrices, and by $O(n^{1.2})$ for $M$-matrices of planar graphs.

Chen and Toledo [30] report that Vaidya's preconditioners seem to converge at an almost constant rate. Moreover, as predicted by the theoretical analysis, they find that such preconditioners are sensitive only to the nonzero structure of the co-efficient matrix, and not to the values of its entries.

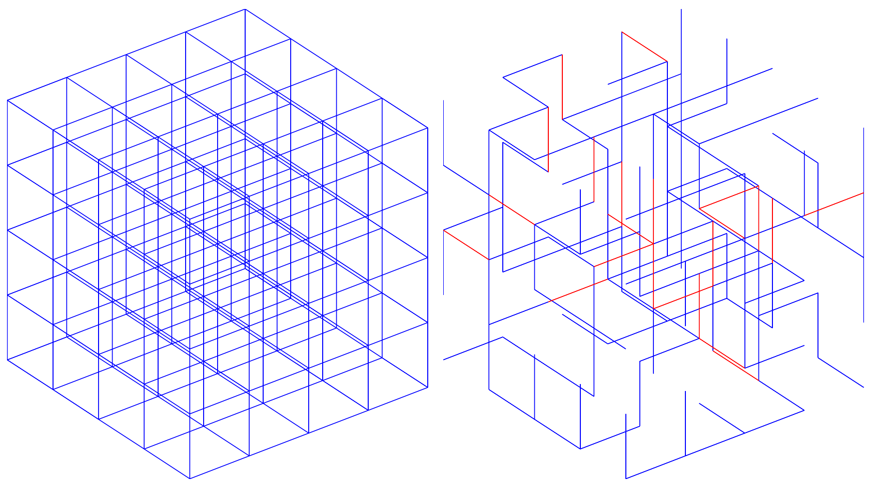


**Figure 5.12:** The 3D model problem and the corresponding Vaidya preconditioner. The basic Vaidya preconditioner is shown in blue; the augmented Vaidya precondtioner includes the extra red edges.

Figure 5.11 shows the 2D (100 grid points) and figure 5.12 shows th3 3D model problem (125 grid points), along with the corresponding Vaidya preconditioners. The blue edges correspond to the MWST, while the red edges are the extra edges for the augmented Vaidya preconditioner. The basic Vaidya preconditioner simply uses the MWST, the implementation of which is described in chapter 2. Our implementation of the augmented Vaidya preconditioners is shown in code fragment 5.13.

Construction of the augmented Vaidya preconditioner starts with the MWST. It then partitions the tree into subgraphs of equal size, and adds back the heaviest graph edges between the subgraphs. We use the partitioning scheme described by Chen and Toledo [30]. This scheme requires a rooted tree. We

```
function T = vaidya_support (G, npart)
% Compute the Vaidya preconditioner for a symmetric M-matrix

% Get the simple Vaidya preconditioner based on the MST
T = mst (G, 'max');

% Partition the tree
part_label = treepart (T, npart);

% If edge weights are not unique, contract may get confused
Gu = unique_edges (G);
[ign E_SN] = contract (Gu, part_label, 'argmax');

% Augment MST with heavy edges between the tree partitions
U = nonzeros (triu (E_SN));
V = nonzeros (triu (E_SN'));
T_aug = sparse ([U; V], [V; U], 1, length(G), length(G));
T = G .* (T | T_aug);

% Preserve row sums in the preconditioner
rowsumsG = sum (G, 2);
rowsumsT = sum (T, 2);
T = T + diag (sparse (rowsumsG - rowsumsT));
```

**Figure 5.13:** The augmented Vaidya preconditioner. Construction of the augmented Vaidya preconditioner starts with the maximum weight spanning tree. The tree is then partitioned and heavy graph edges between partitions are added back.

implement a simple algorithm to compute the parent information in the tree (code fragment 5.14). The `mst` routine from GAPDT only returns the tree as an adjacency matrix without the parent information. Bader et al [35] describes an efficient pointer jumping algorithm to compute a rooted tree when finding connected components. We believe it should be possible to combine their approach with the Boruvka step to compute a rooted MWST efficiently.

Our parent computation is performed by a series of breadth-first searches (code fragment 5.14). The computation starts at the leaves of the tree and proceeds towards the root. We assume that the tree is connected. Tree leaves have only one parent. The algorithm uses breadth first search to find the parents of leaves (lines 16–19). The leaves are then removed from the tree, and parent degrees are updated (lines 22–23).

The parent information is required for tree partitioning (code fragment 5.15). The routine `treepart` divides the tree into connected subtrees with sizes between $n/t$ and $dn/t - 1$ [30]. $n$ is the number of nodes in the graph, $d$ is the maximum degree of a node, and $t$ is the number of desired subtrees. The algorithm accumulates the number of children in subtrees as it traverses up the tree (lines 18–19). If there are more than $n/t - 1$ nodes in a subtree at any given point, the subtree is cut from the tree (lines 22–27). Upon termination, the connected components routine is used to recover the partitions. The optimal

```
function parent = getparent (T)
% Parent information for nodes in a tree

n = length (T);
T = spones(zerodiag (T));
parent = zeros (n, 1);

while true
    % Find the leaves at this level
    outDegree = sum(T,2);
    leaves = find(outDegree == 1);
    leafSize = length(leaves);

    if isempty(leaves) break; end

    % Find the parents of these leaves
    leafMatrix = sparse(leaves, 1:leafSize, 1, length (T), leafSize);
    parentMatrix = T * leafMatrix;
    [connections, ign] = find(parentMatrix);
    parent (leaves) = connections;

    % Drop the edges from the original tree
    S = sparse([connections leaves], [leaves connections], 1, n, n);
    T = T - S;
end
```

**Figure 5.14:** Compute parents of nodes in a tree. The computation starts with the leaves of the tree and works its way up the tree. Leaves are nodes of degree one; they have only one parent. Parents of leaves are located with breadth-first search (implemented with matvec).

```
function partlabel = treepart (T, subgraphs)
% Partition a tree into subgraphs of roughly equal size

splitBound = ceil(n / subgraphs);  % Node Splitting criteria
parent = getparent(T);  % Get parent information
deg = sum(T, 2);        % Node degrees
notSplit = ones(n, 1);  % Nodes which have been processed
nKids = ones(n, 1);     % Number of children

while ~isempty(leaves)
  leaves = find (deg == 1);
  leafparent = parent (leaves);

  % Update number of children in subtrees
  toAdd = sparse (leafparent, 1, nKids(leaves), n, 1);
  nKids(leafparent) = nKids(leafparent) + toAdd(leafparent);

  % Find nodes that can form subtrees (partitions)
  splitRoot = find (nKids >= splitBound & notSplit);
  splitParent = parent (splitRoot);

  % Disconnect partitioned subtrees from the tree
  % Update node degrees
  ...

  notSplit (splitRoot) = 0;
end

% Use connected components to assign labels to partitions.
partlabel = components(T);
```

**Figure 5.15:** Partition a tree into subtrees of roughly equal size. The algorithm accumulates the number of children in subtrees as it traverses up the tree. Subtrees are disconnected if they have sufficient nodes to satisfy the partitioning criteria. Upon termination, the connected components routine is used to recover the partitions. Some lines in this code are omitted for simplicity.

number of partitions may need to be determined experimentally for a specific problem.

### 5.3.2 Algebraic multigrid preconditioners

Multigrid methods [23] are attractive because they can solve linear systems with $N$ unknowns (nodes) typically arising from certain elliptical PDEs in $O(N)$ time. Multigrid methods also parallelize well and have been scaled to hundreds of processors [1].

Multigrid methods solve the system on several different scales, using two complementary processes: smoothing and coarse grid correction. A few iterations of Jacobi or Gauss-Seidel are used to smooth out high frequency error. Coarse grid correction uses a *restriction* operator to restrict the residual to a coarse grid, solves the system on the coarse grid, and then transfers the solution back to the fine grid with an *interpolation* operator. The coarse grid correction eliminates low frequency error.

Algebraic multigrid [23, 116] derives its intuition from geometric multigrid, but in a way that does not require explicit knowledge of the underlying geometry. The series of increasingly coarse problems are computed by graph theoretic techniques. Algebraic multigrid (AMG) does not have linear time guarantees and works well for only certain types of systems.

Both geometric and algebraic multigrid methods can be efficiently implemented in MATLAB and STAR-P, by storing the grids as sparse matrices. Sparse matrix matrix multiplication (chapter 2) is useful in the construction of coarse grids. Sparse matrix dense vector multiplication is used to implement the Jacobi algorithm for relaxation. It is also used for coarse grid correction, to transfer residuals between hierarchies of grids.

The $O(n)$ complexity of multigrid is achieved through the V-cycle, which solves the problem recursively on coarser grids and then interpolates the solution back to finer grids. Code fragment 5.16 describes the V-cycle. In geometric multigrid, the restriction and interpolation operators are computed using the discretization stencil. In AMG, these are computed with graph theoretic techniques.

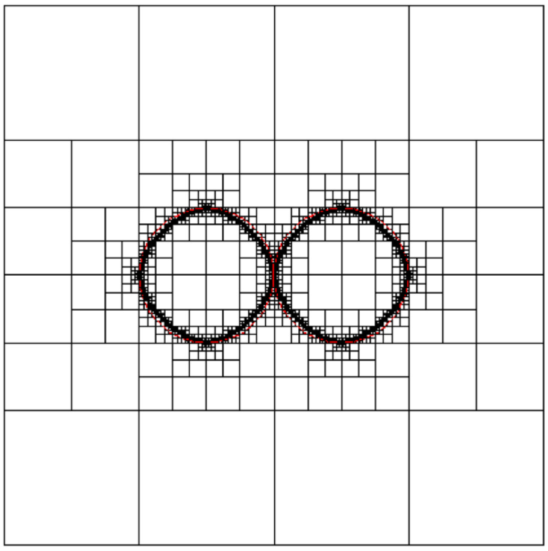
Coarse grid selection first defines a *strength matrix* $A_s$. Weak connections (edges) in $A$ are deleted. A common criterion for picking weak connections is to use a strength parameter $\theta$. Entries smaller than $\theta$ times the largest row entry in the matrix are considered as weak connections and dropped from $A_s$. An independent set of nodes is then selected from the graph $A_s$. Finally, additional points may be selected in a second pass if needed to satisfy interpolation requirements. This second pass is expensive, and has been largely abandoned

```
function z = mgv (operators, b, level)
% The multigrid V-cycle

A = operators.A{level};
I = operators.Interpolate{level};
C = operators.Coarsen{level};

% Do a direct solve if at the bottom of recursion
if level == operators.levels; z = A \ b; return; end

% Relaxation - Typically Jacobi or Gauss-Seidel
x = relax (A, b);

% Coarse grid correction
b_coarse = C * (b - A * x);       % Coarsen the RHS
z = mgv (ops, b_coarse, level+1); % Solve recursively
z = x + I * z;                     % Correct the solution
```

**Figure 5.16:** The multigrid V-cycle.

in favor of other approaches for defining interpolation. The algorithm can be sensitive to the choice of $\theta$.

The independent set algorithm used in AMG differs slightly from the one discussed in chapter 3. When a node is added to the independent set, it is designated as a $C$ point, belonging to the coarse set. Its neighbors are designated as $F$ points, belonging to the fine set. The degrees of the neighbors of $F$ points are incremented to make it more likely for them to be picked next by the independent set algorithm. Our MIS implementation in GAPDT can be modified to use this heuristic for an efficient parallel implementation.



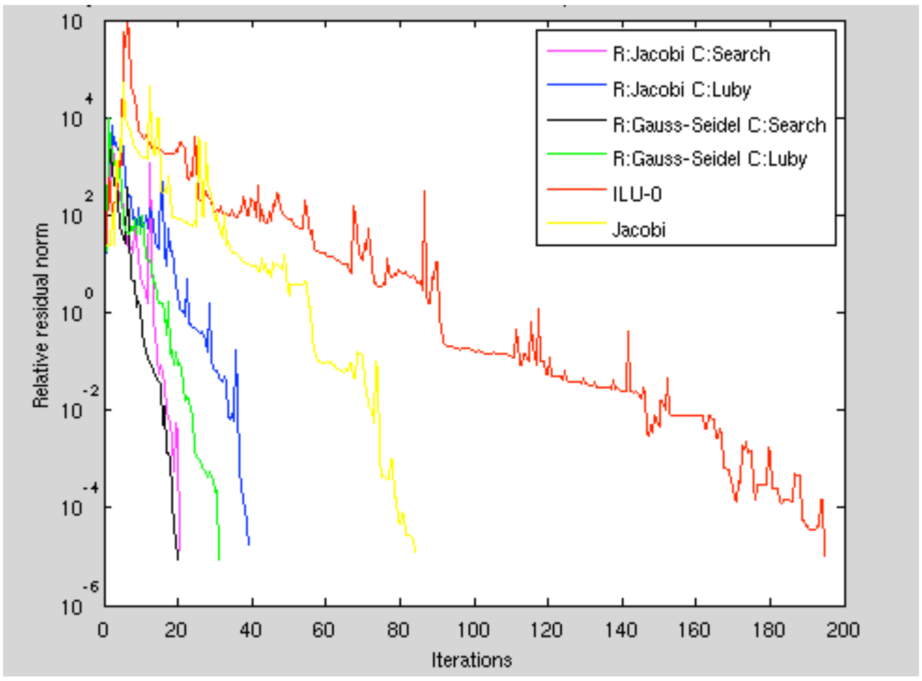
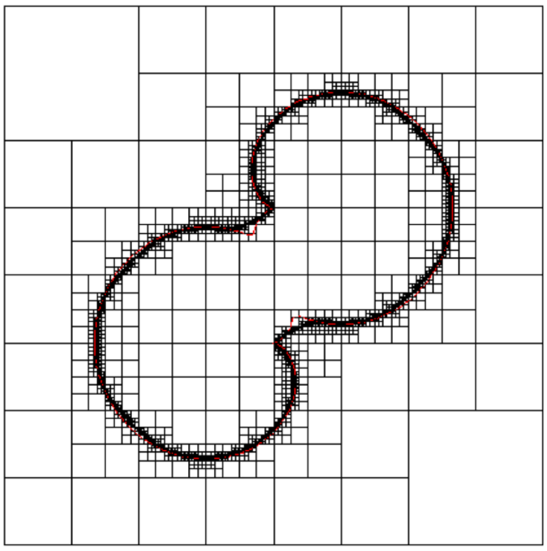


**Figure 5.17:** The top image is a quadtree discretization of a level set problem. Grid Image reproduced with permission from Vikram Aggarwal. The graph shows performance of bi-conjugate gradient with several AMG preconditioners, a Jacobi preconditioner, and an ILU preconditioner.



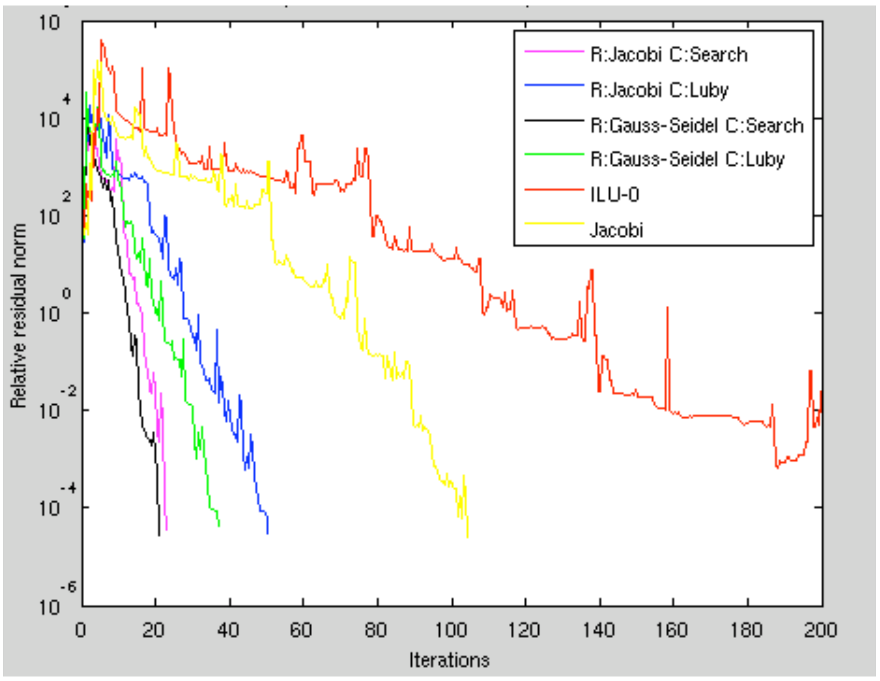


**Figure 5.18:** The top image is a quadtree discretization of a level set problem. Grid Image reproduced with permission from Vikram Aggarwal. The graph shows performance of bi-conjugate gradient with several AMG preconditioners, a Jacobi preconditioner, and an ILU preconditioner.

We describe an implementation of AMG by Roh and Shah [104], to solve a level set problem in fluid dynamics [3, 95]. The quadtree discretization along with a second order accurate method produces unsymmetric matrices. As a result, they use preconditioned BiCGstab [14] to solve the linear systems.

Since the matrix $A$ is unsymmetric, the symmetric matrix $(A + A^T)/2$ is used to compute the coarse and fine grid operators. Figures 5.17 and 5.18 show the meshes arising from quadtree discretizations, and the speed of convergence with several AMG preconditioners, a Jacobi preconditioner, and an ILU precondtioner. The AMG code can run in parallel in STAR-P, except for the computation of coefficients of the interpolation matrix, which we have not yet vectorized. All variants of AMG preconditioners use fewer iterations than Jacobi or ILU preconditioners, even though Jacobi gives the fastest time to solution. We believe an approach based on AMG may be promising to parallelize large problems arising from quadtree discretizations of level set problems. The effectiveness of this method for octree discretizations of 3D problems has yet to be investigated.

## 5.4 Non-negative matrix factorization

Non-negative matrix factorization (NNMF) [82] is a useful tool for machine learning. NNMF is similar to principal component analysis (PCA) and vector quantization (VQ): $V \approx WH$. All three methods are approximate matrix factorizations. The $r$ columns of $W$ form a basis, and the coefficients in $H$ represent linear combinations of the columns of $W$, which approximate columns in $V$. The matrix $V_{m\times n}$ is approximated by a rank $r$ factorization

$$V_{ij} \approx (WH)_{ij} = \sum_{a=1}^{r} W_{ia}H_{aj}.$$

In VQ, each column of $H$ is constrained to be a vector with zeros in all positions except one. PCA, on the other hand, constrains the columns of $W$ to be orthonormal and the rows of $H$ to be orthogonal to each other. Non-negative factorization (NNMF) constrains the entries of $W$ and $H$ to be greater than or equal to zero.

All three methods can be implemented in MATLAB, and scaled to large problem sizes with STAR-P. Here, we describe our implementation of NNMF in STAR-P. The matrix $V$ is often sparse in applications. Sparsity in $V$ may or may not imply sparsity in $W$ and $H$. Lee and Seung [83] describe two multiplicative algorithms for NNMF. A discussion of our implementations of both these algorithms follows.

A cost function quantifies the quality of a non-negative factorization. The simplest measure of cost is the square of the Euclidean distance between $A$ and $B$,

$$||A - B||^2 = \sum_{ij} (A_{ij} - B_{ij})^2.$$

The corresponding multiplicative update rules that guarantee convergence are

$$H_{aj} \leftarrow H_{aj} \frac{(W^T V)_{aj}}{(W^T W H)_{aj}},$$
$$W_{ia} \leftarrow W_{ia} \frac{(V H^T)_{ia}}{(W H H^T)_{ia}}.$$

Another useful measure of cost is the Kullback-Leibler divergence [81]. In this case, the entries in $A$ and in $B$ must sum to 1, and they can be regarded as probability distributions.

$$D(A|B) = \sum_{ij} (A_{ij} \log \frac{A_{ij}}{B_{ij}} - A_{ij} + B_{ij}).$$

The multiplicative rules that guarantee convergence under Kullback-Leibler divergence are

```
function [W, H] = euclidean_update (V, W, H)
% Euclidean update rules for NNMF

Wt = W';
H = H .* (Wt * V) ./ ((Wt * W) * H + eps);

Ht = H';
W = W .* (V * Ht) ./ (W * (H * Ht) + eps);
```

**Figure 5.19:** Multiplicative update rules to minimize the Euclidean distance cost function.

$$H_{aj} \leftarrow H_{aj} \frac{\sum_i W_{ia} V_{ij} / (WH)_{ij}}{\sum_k W_{ka}},$$

$$W_{ia} \leftarrow W_{ia} \frac{\sum_j H_{aj} V_{ij} / (WH)_{ij}}{\sum_k H_{ak}}.$$

Code fragment 5.19 describes the update rules to minimize the Euclidean distance between $V$ and $WH$. Note that machine epsilon is added to all zero entries in the factorization to avoid division by zero. The matrix $V$ is large and sparse, but the factorization is of low rank. The factors $W$ and $H$ may be dense.

Code fragment 5.20 implements of the divergence update rules. We need to be careful in our implementation of these update rules, since the sparsity of $V$ needs to be preserved. The update rules form the product $WH$, which will typically be dense. Since we are only interested in the values of the product $WH$ corresponding to the nonzero positions in $V$, we explicitly work with the nonzeros in the product.

```
function [W, H] = divergence_update (V, W, H)
% Kullback-Leibler update rules for NNMF

[num_ent, num_feat] = size (V);
[V_ent, V_feat, V_val] = find (V);

WH_val = prodWH (W, H, V_ent, V_feat);
V_over_WH = sparse (V_ent, V_feat, V_val ./ WH_val, num_ent, num_feat);

Ht = H';
W = W .* ((V_over_WH) * Ht) ./ (ones(num_ent, 1) * sum(Ht));

WH_val = prodWH (W, H, V_ent, V_feat);
V_over_WH = sparse (V_ent, V_feat, V_val ./ WH_val, num_ent, num_feat);

Wt = W';
H = H .* (Wt * (V_over_WH)) ./ (sum(W)' * ones(1, num_feat));
```

**Figure 5.20:** Multiplicative update rules to minimize the Kullback-Leibler divergence cost function.

```
function WH_val = prodWH (W, H, V)
% Space saving sparse matrix multiplication for NNMF

[V_ent, V_feat] = find (V);

% Find rows of W which contribute to a nonzero in V
W_V_ent = W(V_ent, :);

% Find columns of H which contribute to a nonzero in V
H_V_feat = H(:, V_feat)';

% Inner product formulation for matrix multiplication
WH_val = sum (W_V_ent .* H_V_feat, 2);
```

**Figure 5.21:** The algorithm needs only those nonzeros in the product of W and H that are present in V. This knowledge is used to compute the product efficiently and keep it sparse, which might otherwise be a full matrix.

Code fragment 5.21 describes how the product $WH$ is formed for only those nonzero positions which exist in $V$. The implementation picks a row in $W$ and a column in $H$, the dot product of which would form a nonzero in $V$. The memory requirement is $O(nnz)$, unlike forming the product $WH$, which may need $O(mn)$ memory.

We used our implementation of NNMF to factor the Netflix challenge problem [97]. The Netflix data consists of movie ratings by viewers. Each viewer rates certain movies on a scale of one to five. The goal is to develop an algorithm that is a good predictor of ratings for movies a user might not yet have rated. There are $17,770$ unique viewers who have rated at least one of $480,189$ movies. We report performance for a rank 5 factorization. It takes 50 seconds to perform ten iterations of the update rules to minimize the Euclidean cost function[2]. Minimizing the Kullback-Leibler divergence is considerably more expensive, taking 1150 seconds for ten iterations.

Our initial investigation suggests that NNMF may not be a good method to solve the Netflix prediction problem. However, we effortlessly implemented a new non-trivial algorithm in parallel on a large problem. This is exactly what we envision our infrastructure to enable: rapid, feedback driven algorithm development on realistic datasets.

[2]14 processors of a 16 core shared memory Opteron computer with 64GB of RAM are used.

## 5.5 Conclusion

We implemented a variety of algorithms using the distributed sparse matrix infrastructure in STAR-P and the graph toolbox built on top of it. We described a problem in computational fluid dynamics, our implementation of a graph analysis benchmark, combinatorial preconditioners for solving linear systems, and non-negative matrix factorizations. The graph benchmark is purely combinatorial, whereas non-negative factorization is a numerical algorithm. The support graph and AMG preconditioners, constructed with combinatorial techniques, are used to accelerate iterative methods for solving linear systems.

# Chapter 6

# Landscape Connectivity: An Application in Ecology

## 6.1 Introduction

Dispersal, the movement of individuals among populations is a critical ecological process that can maintain genetic diversity and rescue declining populations [27]. As areas of natural habitat are reduced in size and continuity by human activities, the degree to which the remaining fragments are functionally linked becomes increasingly important. The strength of such linkages is determined by “landscape connectivity”, which despite its intuitive appeal is inconsistently defined. Measures of connectivity differ in their data requirements and information yield, each having their own strengths and weaknesses. If scarce conservation dollars are to be spent effectively, conservation biologists need clear, efficient, and reliable tools relating landscape composition and pattern to important ecological processes.

McRae et al [93] propose a distance metric based on circuit theory, motivated by the fact that conductive materials display properties analogous to landscape connectivity. They adapt electrical network theory to model populations connected by migration, or raster cells connected by individual movement. Equations describing effective conductance across resistive networks are closely related to individual movement probabilities and random walk times, as well as effective migration in analogous networks of subpopulations. Such a model can be useful for conservation planning and for predicting genetic effects of landscape change because of its ability to integrate all possible routes of inter-patch movement simultaneously.

Circuitscape, a tool written earlier in Java [92], and now in MATLAB, implements a model of animal movement and gene flow in heterogeneous landscapes. Solution of this problem requires both combinatorial and numerical algorithms. Even at moderate resolutions, the underlying graphs of habitat maps can be fairly large, depending on the size of the landscape and the species being modeled. The combination of distributed sparse matrices in STAR-P (chapter 2), the graph toolbox (chapter 3), and vectorization allow computations on large landscapes, which were not possible earlier.

## 6.2 Modeling landscape connectivity

We provide an overview of the intuition behind the modeling process as presented by McRae et al. [93].

Habitat cells connected by migration can be represented as graphs [118]. Habitat cells represent nodes, while migration paths represent edges. The edge weights are proportional to numbers of migrants exchanged between a pair of nodes. Landscape characteristics may cause the number of migrants to vary across the graph, resulting in varying edge weights. However, the edges are assumed to be undirected, which implies that dispersal is balanced.

Isolation by distance (IBD) [112] and least cost paths (LCP) [2] are two common measures of connectivity used in the landscape ecology literature. IBD models use Euclidean distance as their connectivity metric. Least cost paths, on the other hand, use optimal routes between samples.

Isolation by Resistance (IBR) takes advantage of the correspondence between effective resistance in resistive networks and random walks on graphs [45, 79]. Effective resistance captures some important properties needed for a good connectivity metric [112]: it increases with distance in one-dimensional conductors, and with the logarithm of the distance in two-dimensional conductors.

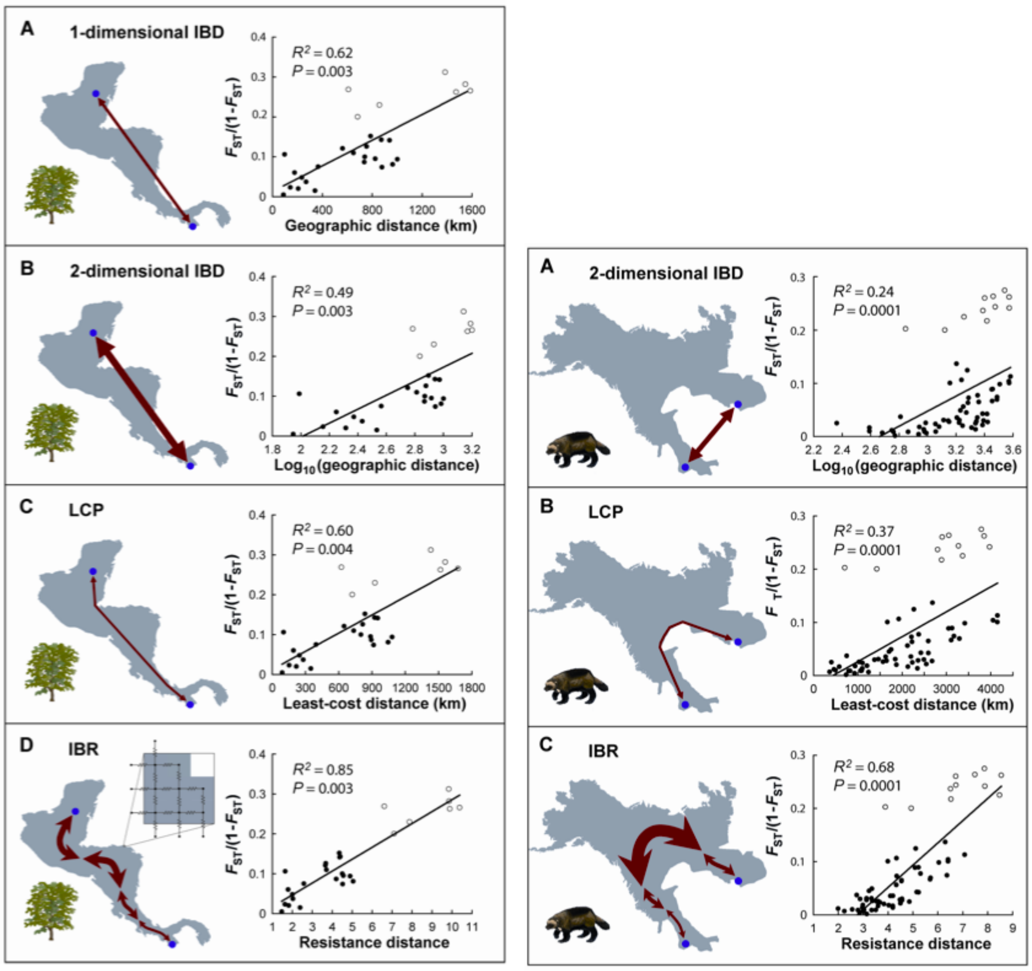


**Figure 6.1:** Comparison of various connectivity metrics for mahogany and wolverine populations. Isolation by distance (IBD) based metrics are poor predictors on landscapes with irregular geometries. Least cost paths (LCP) tend to restrict movement to single, optimal routes. Isolation by resistance (IBR) as a distance works well as a predictor of genetic dispersal between habitat patches in irregularly shaped landscapes. Image reproduced with permission from Brad McRae.

Figure 6.1 shows the application of various modeling techniques to mahogany and wolverine data. IBR modeling provides a much better picture of gene flow in both cases. Unlike IBD models, IBR incorporates effects of limited and irregular habitat extent. Unlike LCP models, IBR also accounts for multiple pathways and wider habitat swaths connecting populations.

Kirchoff's laws [78] are used in matrix form to solve for effective resistances of nodes within circuits. Let $g_{ij}$ denote the conductance of the resistor connecting nodes $i$ and $j$. Let $G$ be an $n \times n$ weighted Laplacian matrix, such that $G_{ij} = -g_{ij}$ and $G_{ii} = \sum_{j=1}^{n} g_{ij}$. Resistance between nodes $x$ and $y$ (with $x < y$ for convenience) may be computed using a reduced conductance matrix $G_y$, which is the same as $G$ but with the $y^{th}$ row and column removed. The right hand side $I$ is a vector with all zeros except in the $x^{th}$ position where it is set to one. Now, solving $G_y v = I$ yields the effective resistance between nodes $x$ and $y$ in the $x^{th}$ element of $v$. The effective resistance is denoted by $\hat{R}_{xy}$, while the effective conductance is its reciprocal and denoted by $\hat{G}_{xy}$.

## 6.3 Computing effective resistance

Circuitscape [92] was developed to apply IBR methods to problems in landscape ecology. The computation typically starts with the raster cell map of a landscape exported from a GIS system. The landscape is exported as a matrix of conductance values assigned to each cell based on landscape features. Every cell in the landscape is represented by a node in the graph. An $m \times n$ cell map results in a graph with $k = mn$ nodes. Neighbor relationships between cells in the landscape are represented as edges in the graph. Edge weights in the graph are an average of the conductance values of the cells they connect. More sophisticated ways of computing edge weights may also be used. The modeling allows a cell to be connected to either 4 neighbors or 8 neighbors.

The size of the graph depends on the size of the landscape and the kind of animal being modeled. The amount of land an animal perceives around itself typically depends on its size: a mountain lion may perceive about a hundred meters of land around it, while smaller animals such as mice may only perceive a few meters [80]. The area of interest may also widely vary. The Yellowstone-to-Yukon [88] (figure 6.2) project is perhaps the most ambitious conservation project ever attempted. It seeks to provide continuous connecting corridors for wildlife movement in protected areas that extend for more than 2000 miles from Yellowstone's hot springs to the Yukon's Mackenzie Mountains.

An ecologist would typically like to work with a resolution fine enough to match the animal being modeled. As a result, graph sizes get large very quickly. For example, a 100 km$^2$ area at 100 meter resolution yields a graph with a

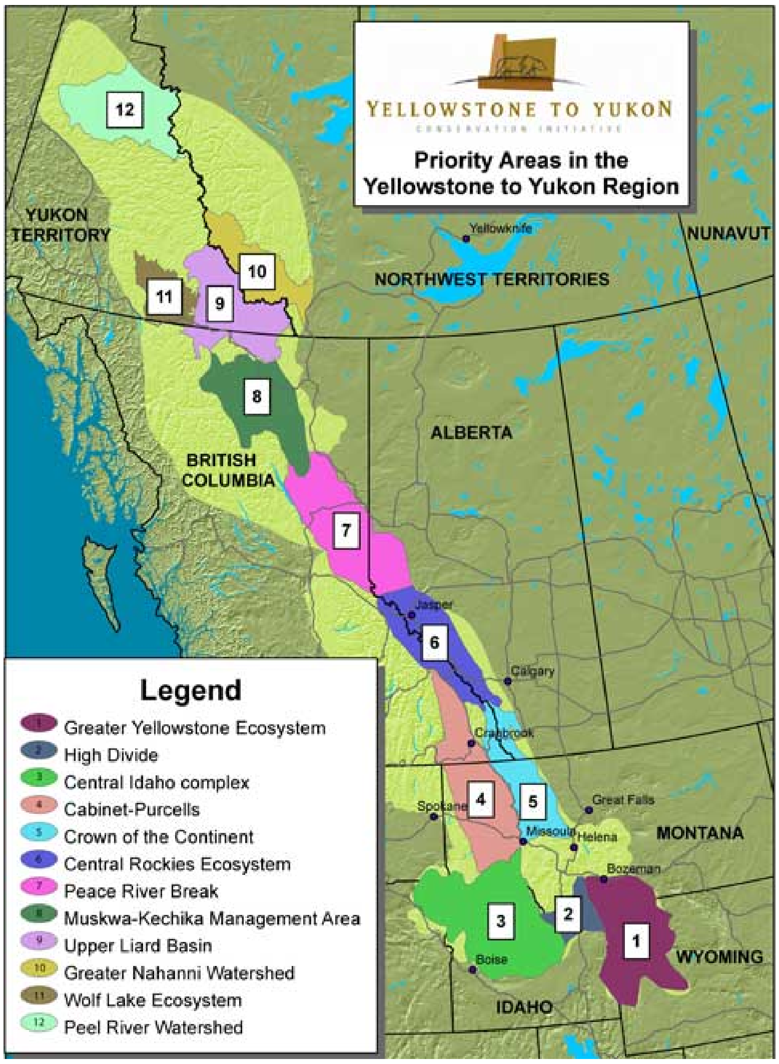


**Figure 6.2:** The ¡¡¡¡¡¡¡ .mine Yellowstone-to-Yukon conservation region (www.y2y.net). The region stretches for ======= Yellowstone-to-Yukon conservation region. The region stretches for ¿¿¿¿¿¿¿ .r276 2000 miles and covers an area of 460, 000 square miles. Image reproduced from www.y2y.net.

million nodes. A landscape that includes the entire state of California would result in a graph with 40 million nodes. Landscapes that span several states can easily result in graphs with hundreds of millions of nodes. The Yellowstone to Yukon region includes $460,000$ square miles of prime wildlife habitat.

We extended the current version of Circuitscape written in Matlab with two basic objectives: it should run sequentially in Matlab for interactive use and development on small to moderate sized problems on desktops. Also, the same code must be able to scale to very large problems on computers with many processors and more memory. We achieved these objectives with a combination of the following:

- We use sparse matrices as a common data structure for combinatorial computations and numerical computations.
- We borrow several concepts and routines from our graph toolbox (GAPDT) for operations on graphs.
- We use preconditioned iterative methods in parallel to solve linear systems for computing effective resistances, since sparse direct solvers lead to unacceptably large fill.

- We vectorized existing code for speed in MATLAB. This also makes it possible to run the same code unmodified in STAR-P. As a result, scaling to large problem sizes is effortless.

The sparse matrix support in STAR-P (chapter 2) and the graph toolbox (chapter 3) were developed specifically for applications such as Circuitscape, which combine combinatorics and numerics to solve a problem. We now discuss the combinatorics and numerics in Circuitscape.

### 6.3.1 Combinatorics in Circuitscape

Circuitscape performs several operations on graphs to obtain the Laplacian of a graph for linear solves. First, it reads the raster cell map from a file and constructs a graph. The raster cell map is represented by an $m \times n$ conductance matrix, where each nonzero element represents a cell of interest in the landscape. Each cell is represented by a node in the graph. Given a node in the graph, the graph construction process inserts an undirected edge connecting the node with its neighbors. There may be up to either 4 or 8 neighbors. As a result, the graph has either 4 or 8 nonzeros per row/column. The choice of neighbor connectivity can affect connectivity of the resulting graph. A graph that is connected with 8 neighbors per cell may not be connected with 4 neighbors per cell.

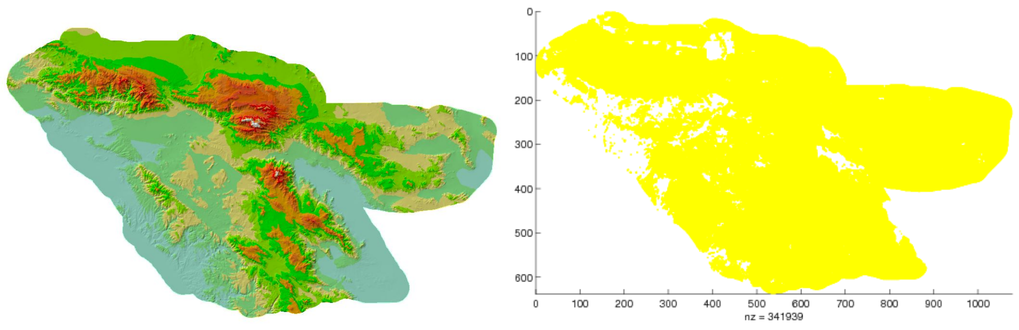


**Figure 6.3:** The southern California region is used as a landscape to model mountain lions. The original raster map from GIS data is shown on the left; MATLAB visualization of the landscape is shown on the right.

The graph construction process is described in code fragment 6.4. The edges in the graph are discovered with stencil operations. The graph is then constructed by calling `sparse()`. Typically, a habitat patch in a landscape will be represented by several nodes in the graph. Since we are interested in effective resistance between habitat patches, all nodes representing a habitat patch must be contracted into one node (figure 6.5 (a)). Neighbors of all nodes of a habitat patch are now neighbors of the contracted node. The resulting graph may also not be connected (figure 6.5 (b)). Using the connected components routine from GAPDT, we drop all nodes that are not in the main component. This node pruning step is necessary in order to avoid singularity in the conductance matrix.

Since we are interested in the current flow between pairs of habitats, nodes representing a single habitat must be contracted into a single node. Depending

```
function G = cs_builder (gmap)
% Generate a graph from a landscape

% Find left-right neighbors
gmap_LR = gmap(:,1:end-1) & gmap(:,2:end);
u_horiz = find ([gmap_LR sparse(nrow,1)]);
v_horiz = find ([sparse(nrow,1) gmap_LR]);

% Find top-bottom neighbors
gmap_UD = gmap(1:end-1,:) & gmap(2:end,:);
u_vert = find ([gmap_UD; sparse(1,ncol)]);
v_vert = find ([sparse(1,ncol); gmap_UD]);

% {s, t} are neighbors. nonzeros in gmap are conductances.
U = [u_horiz; u_vert];
V = [v_horiz; v_vert];
resistances = 1 ./((1./gmap(U) + 1./gmap(V)) / 2);
G = sparse (U, V, resistances);
```

**Figure 6.4:** Stencil operations are used to construct the graph corresponding to a landscape. An edge $u, v$ exists in the graph if and only if cells $u$ and $v$ are neighbors in the cell map.

on the size of the habitat, a habitat node is connected to several other nodes in the graph. This typically results in a dense row and column in the matrix. The actual graph contraction is performed with the `contract` routine from GAPDT.

Since commonly used sequential graph algorithms do not parallelize well, the routines in GAPDT implement efficient parallel graph algorithms and are designed with scalability in mind. This significantly simplifies the implementation of Circuitscape allowing it to manipulate large graphs in parallel with ease.

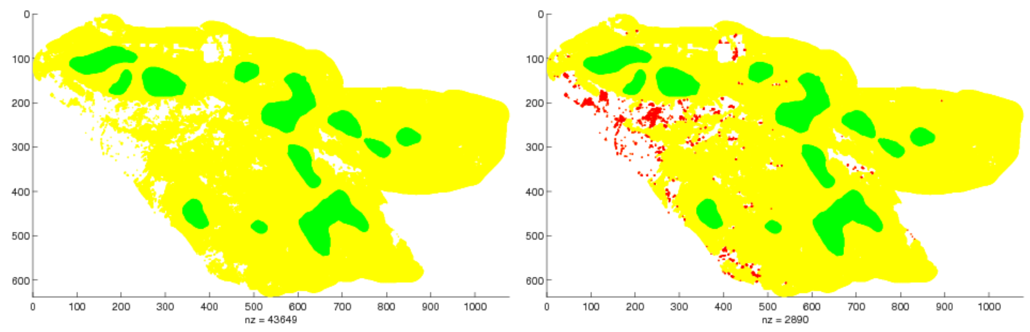


**Figure 6.5:** The landscape graph has to be processed before effective resistance can be computed between habitat patches. First, all habitat patches (polygons), in green, are contracted into one node each (left). Only the main connected component is needed for the effective resistance computation (right). The red parts are disconnected. If the graph includes multiple connected components, the resulting matrix is singular.

```
function G = cs_laplacian (G)
% Generate the Laplacian from the matrix

G = -abs(G);                           % Make off-diagonals negative
G = G - diag(sparse(diag(G)));         % Make diagonal zero
G = G - diag(sparse(sum(G, 2)));       % Make row sums zero
```

**Figure 6.6:** The Laplacian of a graph as computed in MATLAB.

### 6.3.2 Numerics in Circuitscape

Once the graph is constructed, the graph Laplacian is formed as shown in code fragment 6.6. A row and column are then deleted from the Laplacian (making the matrix symmetric positive definite) as described in the section on modeling.

In a typical run, an ecologist might want to compute effective resistance between several pairs of habitats in the graph. In sequential MATLAB, we

| Four neighbors per cell | | | | |
|---|---|---|---|---|
| | With symamd | | Without symamd | |
| Number of cells | Fill | Flops | Fill | Flops |
| $2.5 \times 10^5$ | $6.3 \times 10^6$ | $6.1 \times 10^8$ | $1.4 \times 10^8$ | $7.7 \times 10^{10}$ |
| $1 \times 10^6$ | $3 \times 10^7$ | $5.5 \times 10^9$ | $1 \times 10^9$ | $1.1 \times 10^{12}$ |
| $1.2 \times 10^7$ | $3.9 \times 10^8$ | $2 \times 10^{11}$ | $3 \times 10^{10}$ | $9.5 \times 10^{13}$ |
| $4.8 \times 10^7$ | $1.8 \times 10^9$ | $1.7 \times 10^{12}$ | $2.4 \times 10^{11}$ | $1.5 \times 10^{15}$ |
| Eight neighbors per cell | | | | |
| | With symamd | | Without symamd | |
| Number of cells | Fill | Flops | Fill | Flops |
| $2.5 \times 10^5$ | $1.2 \times 10^7$ | $1.8 \times 10^9$ | $1.5 \times 10^8$ | $8.2 \times 10^{11}$ |
| $1 \times 10^6$ | $6.1 \times 10^7$ | $1.8 \times 10^{10}$ | $1.1 \times 10^9$ | $1.2 \times 10^{12}$ |
| $1.2 \times 10^7$ | $8 \times 10^8$ | $6.8 \times 10^{11}$ | $3 \times 10^{10}$ | $9.7 \times 10^{13}$ |
| $4.8 \times 10^7$ | $3.7 \times 10^9$ | $6 \times 10^{12}$ | $2.4 \times 10^{11}$ | $1.6 \times 10^{15}$ |

**Table 6.1:** Fill-in and flops required to solve linear systems. A fill reducing permutation such as one computed by `symamd` can significantly reduce the space and time required to solve linear systems. However, for large problems, Gaussian elimination is still not feasible.

use the Cholesky factorization of the matrix for linear solves after applying `symamd` [6] for a fill-reducing permutation. We save the Cholesky factor for a given habitat patch while computing the resistance between it and all other habitats.

Consider the $n$ by $n$ symmetric positive definite system of equations $Ax = b$, where $n$ is large and $A$ is sparse. The Cholesky factors of $A$ are computed: $A = R^T R$. Typically, $R$ is much less sparse than $A$, due to fill. However, solving the reordered system $(PAP^T)(Px) = Pb$ can result in much less fill, if $P$ is chosen appropriately [6, 59, 89]. The fill-in limits the largest problem we can solve. We show the fill and floating point operations required for typical

| Four neighbors per cell | | | | |
|---|---|---|---|---|
| Number of cells | Build graph | Contract | Prune | Eff. resistance |
| $250 \times 10^3$ | 1.3 | 1.4 | 4.3 | 6.5 |
| $1 \times 10^6$ | 6.3 | 7.1 | 18.7 | 26.1 |
| $12 \times 10^6$ | 61.3 | 56.9 | 160.3 | 292.2 |
| $48 \times 10^6$ | 355.9 | 310.4 | 822.9 | - |
| Eight neighbors per cell | | | | |
| Number of cells | Build graph | Contract | Prune | Eff. resistance |
| $250 \times 10^3$ | 2.6 | 2.1 | 6.5 | 10 |
| $1 \times 10^6$ | 11.8 | 9 | 30 | 46.3 |
| $12 \times 10^6$ | 149.2 | 113.7 | 325.7 | 620.7 |
| $48 \times 10^6$ | 681.9 | 497.6 | 1353.5 | - |

**Table 6.2:** Time (in seconds) spent in various stages of Circuitscape in sequential MATLAB. The largest problem ran out of memory in the direct solver.

problems that we are interested in solving in table 6.1. The row and column counts for Cholesky factorization can be computed efficiently without actually performing the factorization [57].

Optimal Cholesky decomposition of the 2D model problem on a unit square requires $O(n \log n)$ space and $O(n^{3/2})$ time. Although the space requirements seem modest asymptotically, they are prohibitive in practice for large problems, as shown in table 6.1. The number of floating point operations for the largest problems is also prohibitive. The analysis for the 2D model problem holds for matrices generated from landscapes represented as 2D grids.

For our parallel implementation with STAR-P, we use preconditioned conjugate gradient to solve the linear systems [14]. Conjugate gradient is an iterative method. Its convergence can be sped up significantly by using an effective pre-

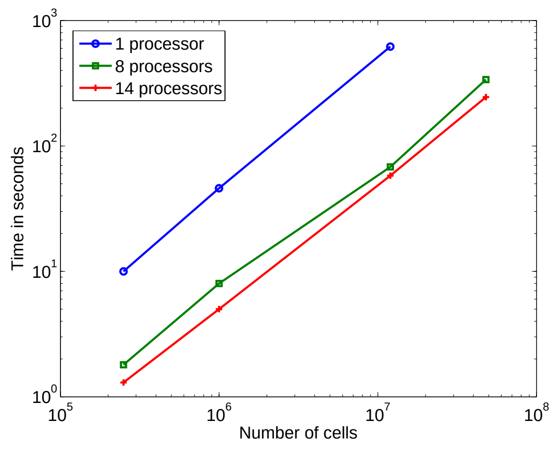


**Figure 6.7:** The effective resistance computation scales well. This is an indicator of the underlying combinatorial preconditioner working well.

conditioner. Instead of solving $Ax = b$, one can solve $M^{-1}Ax = M^{-1}b$. The speed of convergence of conjugate gradient depends on the condition number of $A$. Hence, convergence can be accelerated if $M^{-1}$ has a smaller condition number than $A$. The Hypre library [47] provides several robust high performance iterative methods and preconditioners. We use algebraic multigrid (AMG) as a preconditioner for conjugate gradient. Since AMG is a combinatorial preconditioner, it does not require any extra information about the problem being solved. Hypre includes BoomerAMG [69], a parallel implementation of algebraic multigrid.

| Four neighbors per cell | | | | |
|---|---|---|---|---|
| Number of cells | Build graph | Contract | Prune | Eff. resistance |
| $250 \times 10^3$ | 1.4 | 1.6 | 7.4 | 1.9 |
| $1 \times 10^6$ | 5.2 | 6.0 | 23.9 | 9.0 |
| $12 \times 10^6$ | 43.2 | 52.2 | 185.0 | 76.9 |
| $48 \times 10^6$ | 184.7 | 219.6 | 767.3 | 354.3 |
| Eight neighbors per cell | | | | |
| Number of cells | Build graph | Contract | Prune | Eff. resistance |
| $250 \times 10^3$ | 3.3 | 2.1 | 10.1 | 1.8 |
| $1 \times 10^6$ | 11.5 | 7.4 | 33.1 | 7.5 |
| $12 \times 10^6$ | 114.4 | 68.5 | 283.6 | 68.0 |
| $48 \times 10^6$ | 423.1 | 281.0 | 1193.0 | 338.5 |

**Table 6.3:** Time (in seconds) spent in various stages of Circuitscape in Star-Pwith 8 processors.

## 6.4 Performance of Circuitscape in parallel

We perform a comprehensive performance analysis using the Riverside County landscape (figure 6.3) at several resolutions. Table 6.2 shows the time spent in different stages of the computation within Circuitscape, when running sequentially in Matlab. Tables 6.3 and 6.4 show the performance when the computation is performed in parallel using 8 and 14 processors[1].

The time to solution scales well. Scaling results for the most time consuming step (computing effective resistance) on the largest problem are presented in figure 6.7

[1]All computations were performed on a shared memory 16 core Opteron computer with 64G of RAM.

| Four neighbors per cell | | | | |
|---|---|---|---|---|
| Number of cells | Build graph | Contract | Prune | Eff. resistance |
| $250 \times 10^3$ | 1.3 | 1.7 | 6.9 | 1.2 |
| $1 \times 10^6$ | 3.6 | 5.3 | 18.9 | 5.5 |
| $12 \times 10^6$ | 32.4 | 48.5 | 160.6 | 61.0 |
| $48 \times 10^6$ | 139.9 | 205.4 | 669.6 | 287.5 |
| Eight neighbors per cell | | | | |
| Number of cells | Build graph | Contract | Prune | Eff. resistance |
| $250 \times 10^3$ | 3.2 | 2.2 | 8.7 | 1.3 |
| $1 \times 10^6$ | 8.0 | 6.7 | 26.9 | 4.9 |
| $12 \times 10^6$ | 69.8 | 60.9 | 237.1 | 57.7 |
| $48 \times 10^6$ | 301.3 | 270.9 | 1037.4 | 245.0 |

**Table 6.4:** Time (in seconds) spent in various stages of Circuitscape in Star-Pwith 14 processors.

## 6.5 Conclusion

We described our improvements to Circuitscape, which allow it to solve very large problems. Originally, Circuitscape took a few hours to solve small problems and a few days to solve moderate sized problems (landscapes with approximately 1 million cells). Our speedups result from using the graph toolbox (GAPDT) for graph computations, Star-P for scalability and parallelization, and state of the art linear solvers to compute effective resistance, along with basic vectorization of the rest of the sequential code. Our improvements allow us to solve the largest problems within a few minutes.

We believe that our enhancements will dramatically improve the productivity of users of Circuitscape, allowing them to model landscapes at a resolution fine enough for the results to be realistic. This will lead to a better understanding of landscape ecology — in particular gene flow, and corridor identification for conservation.

# Part II

# Productivity

# Chapter 7

# Design of Experiments in Software Engineering

## 7.1 Motivation

Programming languages have evolved a great deal over the years. It is generally agreed upon that advances in programming language design have led to vast improvements in programmer productivity. There is very little objective evidence to support this claim. This is partly because there is very little data on programmer workflows, and partly because programmer productivity depends upon a number of factors, many of which are hard to quantify. This chapter describes a framework for conducting experiments and the nature of data to be collected.

Designers of computer hardware and software have been concerned about programmer productivity ever since the early years of computing. Backus' optimizing FORTRAN compiler [9] was perhaps the earliest breakthrough to result in a significant improvement of programmer productivity. Designers of programming languages give a lot of thought to programmer productivity, making it a widely studied topic in computer science, if only informally. The discipline of software engineering has studied programmer productivity extensively since the early years of computing [123].

Several studies [15, 61] have repeatedly emphasized the need for experimentation and data collection in measuring programmer productivity. Only with data and a focus on experimentation can we resolve questions about effective design of computer hardware and programming languages. Recognizing the lack of programmer data available to those researching programmer productivity, we are releasing a high quality dataset collected from our classroom experiments. We describe in detail the setup of our data gathering process.

A few datasets on software engineering are available through NASA's defunct Software Engineering Laboratory [16]. Larger samples of data would allow validation of widely held beliefs about programming. Indeed, good quality data collection might even suggest good theories that escape our attention due to lack of data. Often in science, well thought out experiments and careful data collection have led to better theories themselves.

One common hurdle when designing an experiment to gather productivity data is the scope of the data collection and the means by which it should be achieved. Data collection from such experiments needs to meet two conflicting objectives:

1. Collect data on everything that can possibly be measured in order to train models yet to be developed.

2. Keep the data collection process simple and robust without affecting a programmer's workflow.

In our experiments, we balance the two objectives, collecting enough data to allow a researcher to extract unobserved variables, while not distracting the programmer's workflow.

## 7.2 UCSB classroom experiments and replay

We describe two experiments using the Applied Parallel Computing class at UCSB (CS240A), in 2004 and 2006. The class is particularly challenging because students are exposed to a new style of programming. The students learnt to program parallel computers using MPI [44] and UPC [36]. MPI, the Message Passing Interface is a library that allows processors to communicate by exchanging messages. Unified Parallel C (UPC), on the other hand has native parallel constructs.

We believe that instrumentation of any programmer activity should be as non-intrusive as possible. In the 2004 class, we used questionnaires as well as automatic classification. Our positive experience with automatic classification in 2004 encouraged us against using questionnaires in the 2006 class. We discuss the issue of questionnaires vs. automatic classification in section 7.3.

The key feature that distinguishes our classroom experiments at UCSB from other such experiments is the replay. The replay involves rerunning intermediate versions of a programmer's codes to extract information not gathered by the instrumentation tools. In a perfect world, where instrumentation tools are perfect, such replays may be unnecessary. On the other hand, the experiment design can be much simpler if the focus is on ensuring accurate replays of programmer experience at a later time.

There may be a variety of reasons to replay programmer experience. For instance, a researcher may need data which was not gathered by the instrumentation tools. Sometimes, the instrumentation tools may be buggy and may not capture all the data that was intended to be captured. In both our experiments, we found the replay to be a valuable tool. The replay also makes the experiments repeatable, which is a desirable feature for scientific experiments.

### 7.2.1 The 2004 experiment

The 2004 classroom experiment was a pilot. The lessons we learnt from it were used in the 2006 experiment to make the data collection process more robust. In 2004, we were not sure about the kind of data to be collected in such an experiment. Our goal was to collect at least as much data as would be needed to go back and replay each student's experience.

Four homeworks were assigned in this class. All of them were instrumented with the UMD instrumentation tools [127] to capture timestamps, compile times and runtimes. We integrated CVS with the Makefiles provided to the students, so that a source snapshot would be captured every time a student compiled their program.

One of the homeworks the students programmed was a parallel sort. The parallel sort was programmed as a module for an early prototype of STAR-P [76]. As a result, all students used a common harness and common Makefiles. After the class was over, we replayed every single compile for every student in the class and ran it. The consistency provided by the harness make this possible. However, not all run specifics were captured, such as number of processors used, the data generator used by the students, compute time (not total time), and correctness.

We used proxies for run specific parameters. We tested every correct compile on a problem of 10 million elements with 8 processors. The input to the sorting algorithm was a random vector generated by Star-P. All results were compared against Star-P's sort for correctness. We captured the time required to sort the input, not timing Star-P startup time, data generation time or validation time. The replay took one night of computing time on a 32-processor departmental cluster and generated 20 gigabytes of executables.

As a result, we obtained a good insight into the development process the students experienced. It is not the exact student experience, but quite close. One of the benefits of the replay approach using a common input for all runs is that any two runs may be directly compared. Although simplistic, such an approach may give insight into a programmer's thought process and priorities as they develop a program.

### 7.2.2 The 2006 experiment

The goal of our 2006 classroom experiment was to improve upon the methods used in the 2004 experiment and obtain higher quality data. We also wanted to compare two different programming models in this class. Since the replay had proven to be extremely useful in recovering data from the earlier experiment, our 2006 experiment was designed from the ground up with the requirement of allowing an accurate replay.

The main shortcoming of the 2004 experiment was the lack of precise run time parameters used by the students. Our experiments also convinced us that providing well designed harnesses to students enhanced their educational experience, allowing them to focus solely on the computational problem.

The students programmed three homeworks in this experiment. We collected data from two of these. Homework 1 was an introduction and hence it captures novice programmers learning MPI and UPC. The students programmed the power method with dense matrices in MPI and UPC. For homework 2, the students programmed the game of Life. Half the class used MPI and the other half used UPC.

The students were provided with harnesses for every homework. These harnesses were written in C and UPC and provided the following components:

1. Sequential Matlab code
2. Makefiles
3. Supporting code including main()
4. Data generators
5. Validators for specific problems
6. Harness version control

The students were required to program just one function for each homework. Several data generators were provided, with validators for several problems sizes to allow for debugging. The harness version control ensured that all students used the latest available version of the harness, as bugs were fixed.

The students used a commodity cluster to program homework 1. Since this was an introductory homework, full credit was awarded for just getting the right answer. The students developed their game of life codes for homework 2 on 8 processor SMP nodes on Datastar[1], an IBM SP-based computer at the San Diego Supercompute Center. Towards the end of the homework, a challenge problem was announced, which the students had to solve on four 8-way nodes or 32 processors in all.

The harnesses captured all relevant run time information to allow a complete replay if needed (in addition to the UMD instrumentation tools):

1. Number of processors
2. Data generator
3. Problem size
4. Compute time
5. Correctness

[1]Datastar has 176 (8–way) P655+ and 7 (32–way) P690 compute nodes. The 8-way nodes have 16 GB, while the 32-way nodes have 128 GB of memory. DataStar nodes are suitable for both shared-memory and message-passing programming models, as well as the mixture of the two.

It turned out that the compute time was not captured accurately due to a bug in the tools. The replay came to the rescue. The replay took 900 processor hours of compute time on Datastar, generating 20G of executables.

## 7.3 Questionnaires vs. automatic classification of 2004 data

Experiments measuring productivity in the past have relied heavily upon using questionnaires. Questionnaires can be set up in a variety of ways. The classroom studies in HPCS experimented with a variety of questionnaires:

1. **Online Compile-time questionnaires:** Programmers are asked to pick a reason for compiling their code from a list of choices. The 2004 class questionnaire is described in table 7.1.

2. **Offline hourly questionnaires:** These rely on the observation that the transitions between high level programming states (such as debugging, optimization) occur at a much lower frequency than compiles/runs. Programmers simply maintain an hourly log of their activities.

3. **Offline daily questionnaires:** These are similar to the hourly questionnaires, but programmers are asked to keep a log of their activities daily.

The kind of questions asked, and the detail in which they should be answered vary greatly depending on the design of the experiment.

We believe that we get the most accurate data if the programmer is not burdened with extra tasks. Anything that a programmer might not do typically while programming is a distraction, and can affect the quality of data being gathered. Ideally, data collection tools should be completely non-intrusive.

This approach shifts the burden of data collection from the programmer to the experiment designer. The experiment designer must ensure that all the right kinds of data are gathered during the course of the experiment to allow for any subsequent modeling purposes.

Our 2004 experiment used both questionnaires and automatic data collection. The questionnaire posed a multiple-choice question at every program compile. The students picked the activity that most accurately described the task they were performing.

Table 7.1 gives the questionnaire's multiple choice possibilities as well as the heuristics that our automatic deduction used to guess each choice. More than one heuristic might match at any point in time. For instance, a programmer may get a compile time error while trying to parallelize their code, or a run time error when tuning their code.

| **Questionnaire choice** | **Heuristics for deduction** |
|---|---|
| Experimenting with compiler | If the lines of code remains constant between versions and the number of MPI calls is less than 10, the programmer is experimenting with the compiler. |
| Adding serial functionality | If there are still fewer than 10 MPI calls but the number of lines of code changes between versions, |
| Parallelizing | If the number of MPI calls changes between versions, the programmer is adding sequential functionality. |
| Performance tuning | If the running time decreases from one version to the next, it is safe to conclude that the programmer is optimizing the program. |
| Compile time error | If a compile fails, than it is a compile time error. |
| Run time error | If the program crashes or gives an incorrect result, it is a run time error. |

**Table 7.1:** The compile-time questionnaire and heuristics used to automatically deduce reasons for every compile.

We now make some simple comparisons between programmer reported data and automatically classified data. This particular experiment gathered data from 11 programmers. For the purpose of this study, we will assume that a programmer remains in the same state if they report the same activity repeatedly for some period of time.

The simplest test checks for programmers who did not report anything; that is, they reported the same reason for every single compile. It turns out that 3 out of the 11 programmers in the study reported that they were always in the same state through the duration of the experiment. In fact 7 out of the 11 programmers report to have changed states 10 times or less. However, looking

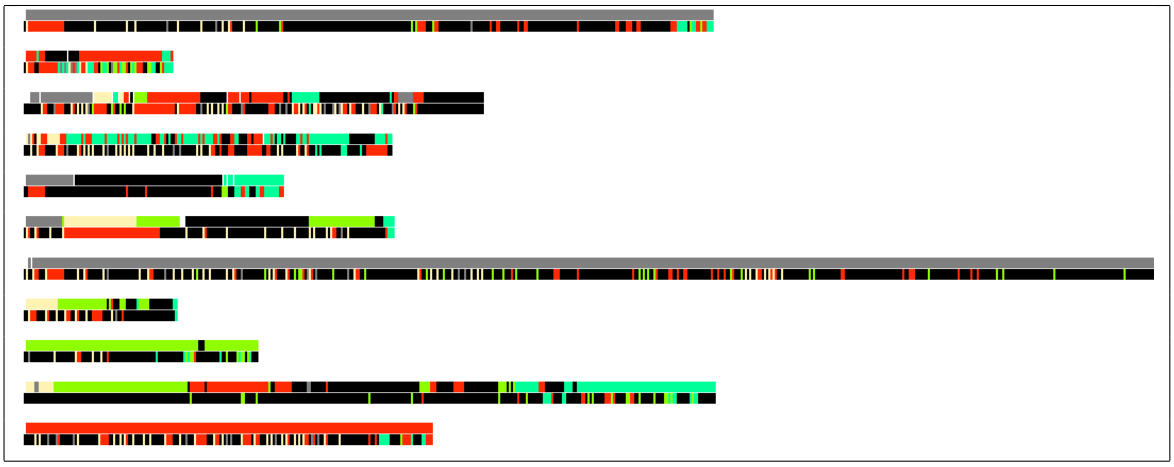


**Table 7.2:** Self-reported versus automatic questionnaires.

at the data, all programmers have had more than 20 compile time and run-time errors. Clearly, there is a mismatch between the reported and the automatically classified data.

We also notice that programmers were either not in a position to guess their true state, or else guessed it incorrectly. Two examples from the data help illustrate this claim.

1. From the data, programmer 10 reported all the state changes very carefully. Despite this, at timestamp 1082800983, the programmer reported "Runtime error" as the reason for the compilation. However, looking at the run time data, we notice that at the previous compile, at timestamp 1082800923, the compile itself failed. Only one line was changed since then, and then the compile succeeded. There was a minor error with the

source code which caused a failed compilation. The correct reason for the compile at timestamp 1082800983 is this failed compilation, but the student chose to report it as a runtime error, even though the code could not be run. This is an example of a programmer misjudging the true state.

2. For programmer 4, at timestamp 1082886139, the reported reason for the compile was "Performance Tuning" even though the programmer did not get any correct run previously. This compile was the 21st compile for the programmer, out of a total of 173 compiles, and thus occurred early in the workflow. The correct reason for this compile could be "Run time error" or "Adding serial functionality". This is an example of a situation in which the programmer did not have the required insight to judge the true state. At the time of development, the programmer could not have known that there would be 173 compiles in total. This makes it difficult for the programmer to judge the true activity taking place. At the end of the development, a complete record of the developer workflow allows us to infer the true state using the overall information, something that the programmer could not possibly do.

There is also the larger issue of different programmers interpreting the meaning of the questionnaires differently. An automatic classification method uses the same heuristics for all data, so that cross–programmer comparison is not only possible but also meaningful.

## 7.4 Visualization and observations from 2006 data

We report some of the data from the classroom experiments, and point out some interesting features in the data. Some observations agree with our intuition, while others do not have an intuitive explanation. These observations are made possible through our detailed data collection and the replay facility.

The figures in tables 7.3 and 7.4 are a visualization of the data collected in the first program in 2006. This is the first time these programmers were exposed to a parallel programming language. Hence, these workflows should be considered to be novice programmer workflows. All programmers used both, UPC and MPI. We only report data for those programmers who correctly implemented the program in both programming languages.

The visualization represents student activity segmented into thirty minute chunks. For each chunk, the color denotes the best activity observed during that interval. The ranking of the activities is as follows: successful run (green) > failed run (orange) > successful compile (yellow) > failed compile (gray).

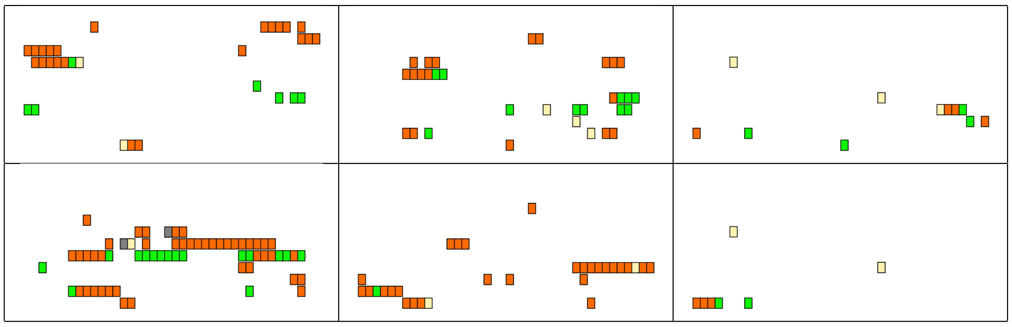


**Table 7.3:** MPI and UPC activities in half hour intervals (students 1, 2, 4). Each half hour interval is represented by a box. The ordering of activities is as follows: successful run (green) > failed run (orange) > successful compile (yellow) > failed compile (gray).

Often programmers use incremental compilation as a tool to ensure that their programs compile, even though there isn't enough functionality implemented for a run. The starting and stopping times are common across all programmers, allowing direct comparison of any pair of workflows.

The first striking observation is that the amount of effort is asymmetric across programming languages for all programmers. Some programmers seem to have spent more time working with MPI, others with UPC. This may be expected since experiences with the first language may allow reuse of knowledge and code for the second language. Novice programmers seem to prefer MPI to start writing their first parallel programs. There could be any number of reasons for this bias; one of them being the availability of plenty of documentation for MPI, and very little for UPC. We are unable to draw any concrete conclusions due to the small sample size, but it is clear that the programmers react

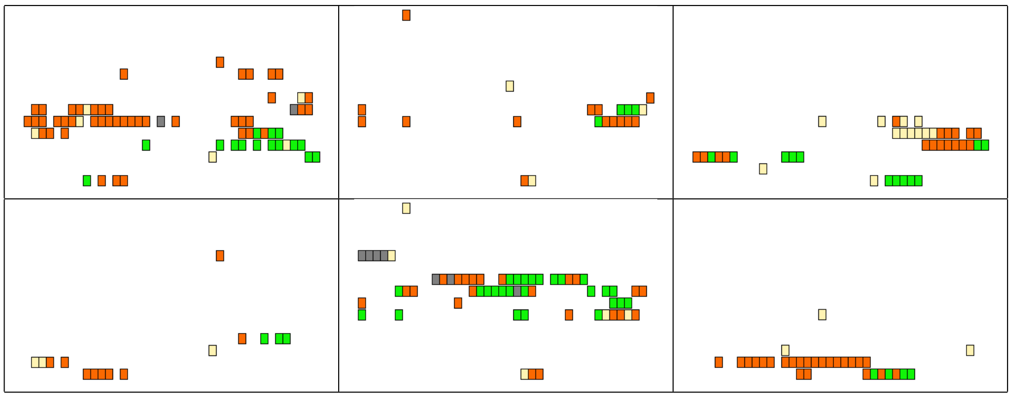


**Table 7.4:** MPI and UPC activities in half hour intervals (students 5, 7, 10). Each half hour interval is represented by a box. The ordering of activities is as follows: successful run (green) > failed run (orange) > successful compile (yellow) > failed compile (gray).

differently to the two languages. For instance, it seems that programmers 1 and 7 expended more effort on UPC, while programmer 5 expended more effort on MPI and programmers 2, 4 and 10 spent an equal amount of effort on both languages.

The programmers were only required to get their programs to run correctly for this assignment. Even though performance was not their primary target, we observe some interesting performance differences across programmers and languages. Table 7.5 reports the performance for successful programmers. There is one concrete observation to be made from this table. Peak UPC performance is comparable to peak MPI performance, but the variance in MPI performance is much larger than UPC performance. The fact that three programmers achieve

| | MPI | | UPC | |
|---|---|---|---|---|
| Student | Min time | Max time | Min time | Max time |
| 1 | 44.3 | 597 | 1020.7 | 1030 |
| 2 | 153 | 186.7 | 349.7 | 349.7 |
| 4 | 1 | 243 | 53 | 67 |
| 5 | 168.6 | 1679.3 | 55.7 | 56.1 |
| 7 | 1004.9 | 1004.9 | 994.7 | 994.7 |
| 10 | 46.8 | 156.1 | 56.7 | 57.3 |

**Table 7.5:** Performance of every successful programmer on assignment 1 in 2006.

close to the best performance in UPC suggests that UPC code does not need to be optimized as much as MPI code.

## 7.5 Conclusion

We provide a generalized framework for experiments to measure programmer productivity. Our experimental design relies on simple, effective instrumentation tools, and captures as much data as is needed to allow for accurate replay. The replay allows a researcher to gather data specific to the model they are working with. Repeatability is a key ingredient of a scientific experiment. We achieve repeatability through replay. We also release the data gathered from our classroom experiments in order to address the lack of good quality data, and hope this will encourage other researchers to do the same.

# Chapter 8

# Timed Markov Models of Programmer Productivity

## 8.1 Introduction

One of the metrics programmers care about when solving a problem on a computer is the time to solution. The clock starts ticking from the moment a programmer starts thinking of the problem and stops when problem is solved. For the sake of simplicity, we consider three major phases in any problem solving process: formulation, programming and execution. The solution can be obtained faster by reducing the time spent in any of these phases. The formulation phase occurs before any code is written. It captures the creative thought process, and again, for simplicity, we will assume it to be independent of the programming and execution phases. Program execution is very much a function of the computer used to run the program. Traditionally, this has been the focus of much research and development in high performance computing. Increasingly, more attention is being focused on the programming phase.

Higher level languages such MATLAB are often thought to be more productive to program in than Fortran or C. Partitioned global address space (PGAS) languages are believed to be easier to use than message passing environments. There are several other widely held beliefs about programming and productivity: a well designed IDE might be more productive than command line tools, a debugger might be more productive than debugging by printing, interactive programming environments might allow for quicker development than compiled languages, and so on.

Such hypotheses are often anecdotal; it is often hard to either prove them or disprove them. It should be possible to confirm or refute hypotheses about programmer productivity with a reasonable model of programming workflows coupled with experimental evidence. Quantitative analysis is desirable, but very hard to get.

We believe that our work can lead to an ability to choose a programming environment based on quantitative evaluations instead of anecdotal evidence. Several studies [28, 70, 99] compare different software engineering processes and programming environments in a variety of ways, mostly qualitative.

Programmers go through an identifiable, repeated process when developing programs, which can be characterized by a *directed graph workflow.* Timed Markov models or timed Markov processes (TMM) are one way to describe such directed graphs in a quantifiable manner. We describe a simple TMM which captures the workflows of. Using our model and tools, we compare the workflows of graduate students programming the same assignment in C/MPI [44] and UPC [36]. This would not be possible without a quantitative model and measurement tools.

## 8.2 Timed Markov processes

The process of software development iterative and probabilistic. It is iterative in the sense that a programmer often repeats a sequence of steps in the software development process: edit, compile, launch test run, for example. It is probabilistic in that the times in each of the steps of the process can vary, and the number of times a cycle will be repeated before the programmer can move on to the next phase is unpredictable. A timed Markov process can model both aspects. Smith, Mizell, Gilbert and Shah describe the first application of such models to software development [114]. Funk, Gilbert, Mizell and Shah report further development of the modeling tools [53].

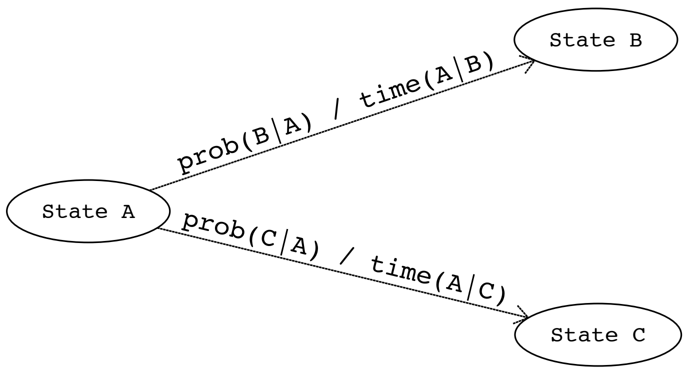


**Figure 8.1:** A timed Markov process

A timed Markov process is a Markov process, augmented with dwell times for state transitions. Each state transition has associated with it, a probability of transition, and a dwell time. Timed Markov processes closely resemble signal flow graphs [71], for which well known methods exist to estimate time to completion. Iterative design processes and software development processes have been studied using similar techniques [77, 115].

In Figure 8.1, $prob(B|A)$ is the probability of transitioning to state B given that the process is now in state A. $time(A|B)$ is the dwell time spent in state A given that the next state transitioned to is B. $prob(C|A)$, which would equal $1 - prob(B|A)$ in this example, is the probability of transitioning to state C given that the process is now in state A.

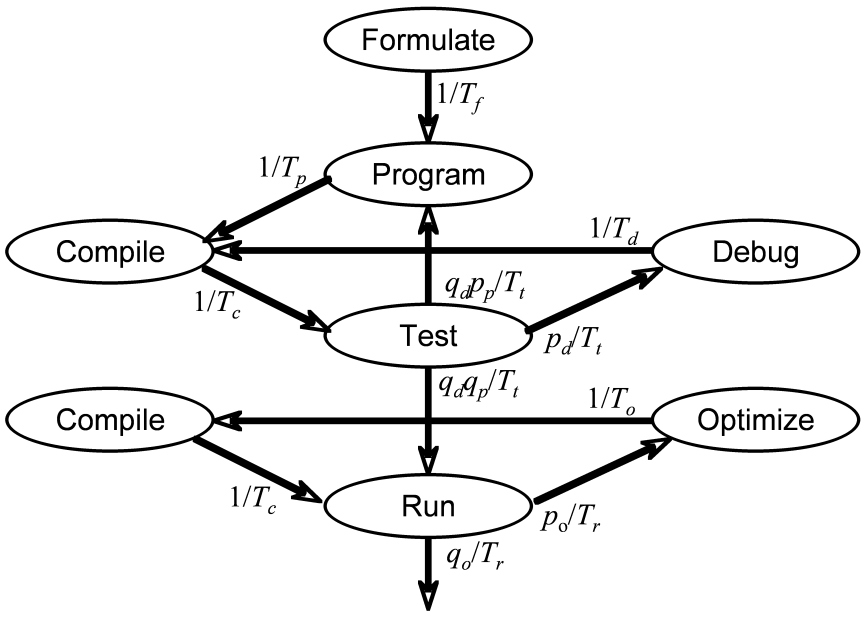


**Figure 8.2:** Lone programmer workflow

## 8.3 Timed Markov models of programmer workflows

In our earlier work [114], we hypothesize a simple timed Markov model of a researcher developing a new application. It represents our assumption that the researcher begins by formulating a new algorithm for solving the problem of interest, and then writes a program implementing the algorithm. Following that, the researcher enters a correctness-debugging loop, around which the process cycles until the program is deemed to be free of programming errors. Next is a performance-tuning loop, which cycles until the program has been tuned enough that it gets adequate performance for large problems to be run on the HPC system. This is the workflow shown in Figure 8.2. All times in this model may be random variables:

- $T_f$ is the time taken to formulate a new approach.
- $T_p$ is the time necessary to implement the new algorithm in a program.
- $T_c$ is the compile time.
- $T_t$ is the time necessary to run a test case during the debugging phase.
- $T_d$ is the time the programmer takes to diagnose and correct the bug.
- $T_r$ is the execution time for the performance tuning runs.
- $T_o$ is the time the programmer takes to identify the performance bottleneck and program an intended improvement.
- $p_p$ is the probability that debugging reveals a need to redesign the program.
- $p_d$ is the probability that more debugging is necessary.
- $p_o$ is the probability that more performance optimization is necessary.
- $q_p$, $q_d$, and $q_o$ are $1p_p$, $1p_d$, and $1 - p_o$, respectively.

This model can also be used to describe the workflow of graduate students programming homeworks in a parallel computing class. In our 2004 classroom experiment, we instrumented the programs and collected workflow data as described in chapter 7. We then fit the collected data to a TMM. While fitting the experimental data to the model, we discovered that in addition to the transitions described above, there were two more transitions. A programmer may introduce or discover a bug while attempting to optimize the program. As a result, there is a transition from the `Run` state to the `Debug` state. Sometimes, the final run may not be successful, perhaps because of a failed attempt to optimize. In such a case, an earlier correct run is treated as the final program (all the data we present is from programs that were eventually correct). Hence, there is another transition from `Test` to `Finish`. Figure 8.3 shows the result of our analysis of the 2004 experimental data.

## 8.4 Comparing UPC and C/MPI workflows

Figure 8.4 shows the workflow of UPC programmers. Figure 8.5 shows that of C/MPI programmers. These diagrams of the TMMs were prepared using Mizell's TMM tool [53]. This is a preliminary analysis with a small sample size (five programmers using each language). Thus we do not attempt to draw final

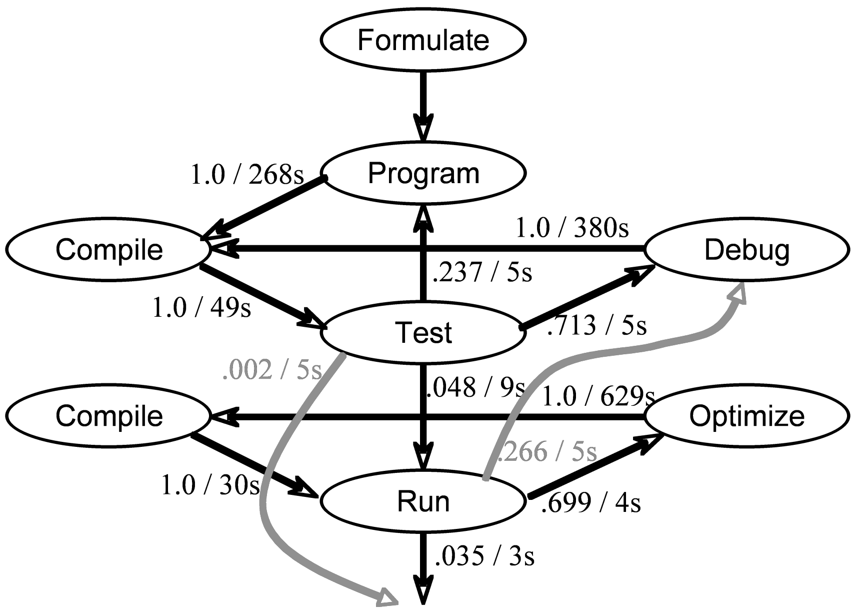


**Figure 8.3:** Fitting data from the 2004 classroom experiment to a timed Markov model.

conclusions comparing the two languages. However, a number of aspects of these TMMs seem encouraging as regards the feasibility of this type of quantitative analysis.

First, the fitted transition times and probabilities from the 2006 classroom experiment are quite similar to those from the 2004 classroom experiment. Not surprisingly, most (92% to 95%) "test" runs lead back into the debug cycle. We see that a "test" run is successful 8% of the time for C/MPI and 5% of the time for UPC; however, in the optimization cycle, 28% of C/MPI runs introduced new bugs compared to only 24% of UPC runs. It is not clear whether these

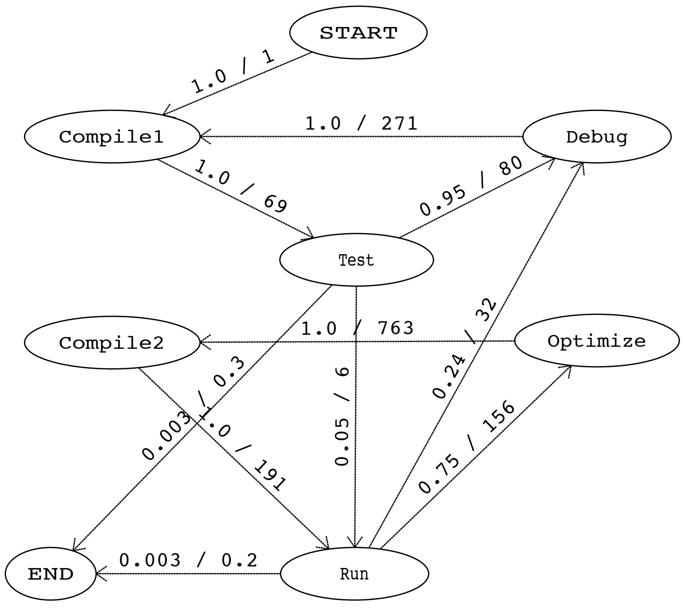


**Figure 8.4:** TMM fit to UPC workflow data. Edges representing state transitions are labelled as: probability of transition / dwell time in seconds.

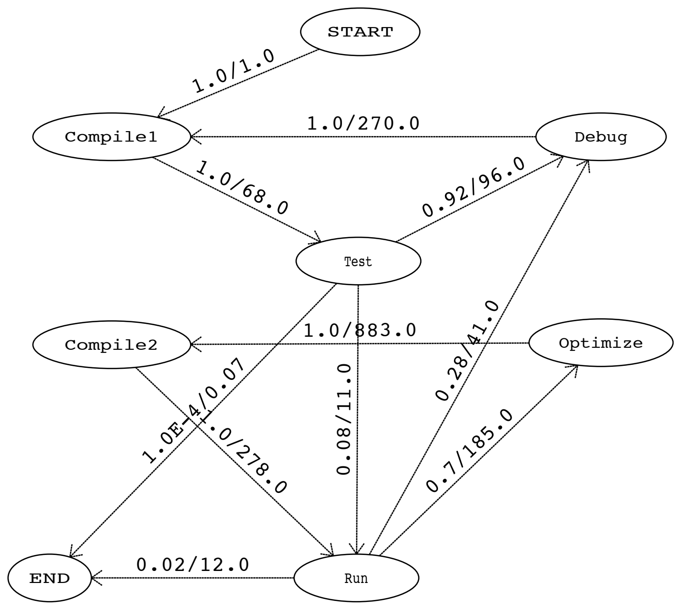


**Figure 8.5:** TMM fit to C/MPI workflow data. Edges representing state transitions are labelled as: probability of transition / dwell time in seconds.

differences are significant for this small sample size. A programmer spends much longer to attempt an optimization (763 seconds for UPC and 883 seconds for C/MPI) than to attempt to remove a bug (270–271 seconds). The time to optimize UPC (763 seconds) is smaller that for MPI (883 seconds), suggesting perhaps that UPC optimization is carried out in a more small-granularity, rapid-feedback way.

## 8.5 Conclusion

We believe that programmers go through an identifiable, repeated process when developing programs, which can be characterized by a directed graph model such as timed Markov models.

We successfully gathered data and fit it to timed Markov models twice, in our 2004 and 2006 classroom experiments. The replay of programmer experience offline was one of the most important aspects of the data gathering process. The timed Markov models clearly indicate the most time intensive parts of the development cycle, quantitatively confirming our intuition — programmers spend most of their time debugging, and once they get a correct program, tuning it for performance is even more difficult. Our data also suggests, in this context, that programmers may introduce slightly fewer bugs in UPC programs, and find it easier to optimize them, as compared to C/MPI programs.

Clearly, this is only the beginning. A lot more data needs to be collected before any comparisons can be in a meaningful way. Towards this end, we are building various tools for the community at large. These tools will provide a general framework for data collection and model construction to study programmer productivity in a variety of ways.

# Bibliography

[1] M. Adams and J. W. Demmel. Parallel multigrid solver for 3d unstructured finite element problems. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 27, New York, NY, USA, 1999. ACM Press.

[2] F. Adriaensen, J. P. Chardon, G. D. Blust, E. Swinnen, S. Villalba, H. Gulinck, and E. Matthysen. The application of 'least-cost' modelling as a functional landscape model. *Landscape and Urban Planning*, 64(4):233–247, Aug 2003.

[3] V. Aggarwal, F. Gibou, and J. R. Gilbert. Effective combinatorial preconditioners for non-graded finite difference octree grids. Technical report, UC Santa Barbara, 2007.

[4] A. V. Aho and J. E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[5] P. Amestoy, I. S. Duff, and J.-Y. L'Excellent. Multifrontal solvers within the PARASOL environment. In *PARA*, pages 7–11, 1998.

[6] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM Journal of Matrix Analysis and Applications*, 17(4):886–905, 1996.

[7] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[8] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, 36(10):1258–1263, 1987.

[9] J. W. Backus. The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computing Conference*, pages 188–198, 1957.

[10] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse. HPCS scalable synthetic compact applications #2. version 1.1.

[11] D. A. Bader. High-performance algorithm engineering for large-scale graph problems and computational biology. In S. E. Nikoletseas, editor, *WEA*, volume 3503 of *Lecture Notes in Computer Science*, pages 16–21. Springer, 2005.

[12] D. A. Bader, K. Madduri, J. R. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch*, Nov 2006.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. *The International Journal of Supercomputer Applications*, 5(3):63–73, Fall 1991.

[14] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. 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, 2nd Edition*. SIAM, Philadelphia, PA, 1994.

[15] V. R. Basili. The role of experimentation in software engineering: past, current, and future. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 442–449, Washington, DC, USA, 1996. IEEE Computer Society.

[16] V. R. Basili, F. E. McGarry, R. Pajerski, and M. V. Zelkowitz. Lessons learned from 25 years of process improvement: the rise and fall of the NASA software engineering laboratory. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 69–79, New York, NY, USA, 2002. ACM Press.

[17] M. Bern, J. R. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo. Support-graph preconditioners. *SIAM Journal of Matrix Analysis and Applications*, 27(4):930–951, 2006.

[18] M. Bern, J. R. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo. Support-graph preconditioners. *SIAM Journal of Matrix Analysis and Applications*, 27(4):930–951, 2006.

[19] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[20] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *Proceedings of the third annual ACM symposium on Parallel algorithms and architectures*, pages 3–16. ACM Press, 1991.

[21] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–460, August 1973.

[22] E. G. Boman and B. Hendrickson. Support theory for preconditioning. *SIAM Journal of Matrix Analysis and Applications*, 25(3):694–717, 2003.

[23] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial: second edition*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[24] G. S. Brodal and R. Fagerberg. Funnel heap – A cache oblivious priority queue. In *Proceedings of the 13th International Symposium on Algorithms and Computation*, pages 219–228. Springer-Verlag, 2002.

[25] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In L. Arge, G. F. Italiano, and R. Sedgewick, editors, *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics*, pages 4–17. SIAM, 2004.

[26] S. V. Browne, J. J. Dongarra, S. C. Green, K. Moore, T. H. Rowan, and R. C. Wade. *Netlib Services and Resources*. Oak Ridge, TN, USA, 1994.

[27] J. M. Calabrese and W. F. Fagan. A comparison-shopper's guide to connectivity metrics. *Frontiers in Ecology and the Environment*, 2(10):529–536, Dec 2004.

[28] B. L. Chamberlain, S. J. Deitz, and L. Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, 2000.

[29] T. F. Chan, J. R. Gilbert, and S.-H. Teng. Geometric spectral partitioning. Technical Report CSL-94-15, Palo Alto Research Center, Xerox Corporation, 1994.

[30] D. Chen and S. Toledo. Vaidya's preconditioners: Implementation and experimental study. *Electronic Transactions on Numerical Analysis*, 16:30–49, 2003.

[31] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam. Algorithm 8xx: Cholmod, supernodal sparse cholesky factorization and update/downdate. Technical Report TR-2006-005, University of Florida, 2006. Submitted to ACM Transactions on Mathematical Software.

[32] R. Choy and A. Edeleman. Parallel Matlab survey, 2001.

[33] R. Choy and A. Edelman. Parallel MATLAB: doing it right. *Proceedings of the IEEE*, 93:331–341, Feb 2005.

[34] E. Cohen. Structure prediction and computation of sparse matrix products. *Journal of Combinatorial Optimization*, 2(4):307–332, 1998.

[35] G. Cong and D. A. Bader. The Euler tour technique and parallel rooted spanning tree. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, pages 448–457, Washington, DC, USA, 2004. IEEE Computer Society.

[36] T. U. Consortium. *UPC Language Specifications V1.2*, May 2005.

[37] T. T. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, 1990.

[38] T. A. Davis. Algorithm 832: Umfpack v4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

[39] T. A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[40] W. L. DeLano and S. Bromberg. *The PyMOL User's Manual*. DeLano Scientific LLC, San Carlos, CA, USA., 2004.

[41] A. V. der Sluis and H. A. V. der Vorst. The rate of convergence of conjugate gradients. *Numerische Mathematik*, 48(5):543–560, 1986.

[42] J. Dongarra, J.R.Bunch, C.B.Moler, and G.W.Stewart. *LINPACK User's Guide*. SIAM, Philadelphia, 1979.

[43] J. J. Dongarra. Performance of various computers using standard linear equations software in a FORTRAN environment. *SIGARCH Computer Architecture News*, 16(1):47–69, 1988.

[44] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, 1996.

[45] P. G. Doyle and J. L. Snell. Random walks and electrical networks. *Mathematical Association of America*, 1984.

[46] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae*, 1959.

[47] R. D. Falgout, J. E. Jones, and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. Design document from the hypre homepage.

[48] M. Fiedler. A property of eigenvectors of non-negative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal*, 25:619–632, 1975.

[49] L. Fleischer, B. Hendrickson, and A. Pinar. On identifying strongly connected components in parallel. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 505–511, London, UK, 2000. Springer-Verlag.

[50] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[51] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[52] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, Los Alamitos, CA, USA, 1999. IEEE Computer Society.

[53] A. Funk, J. R. Gilbert, D. Mizell, and V. Shah. Modelling programmer workflows with timed Markov models. *Cyber Technology Watch*, 2006.

[54] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In D. Kranzlmüller, P. Kacsuk, and J. Dongarra, editors, *PVM/MPI*, volume 3241 of *Lecture Notes in Computer Science*, pages 97–104. Springer, 2004.

[55] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal of Scientific Computing*, 19(6):2091–2110, 1998.

[56] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[57] J. R. Gilbert, E. G. Ng, and B. W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM Journal of Matrix Analysis and Applications*, 15(4):1075–1091, 1994.

[58] J. R. Gilbert, S. Reinhardt, and V. Shah. An interactive environment to manipulate large graphs. In *Proceedings of the 2007 IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 2007.

[59] J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, 50(4):377–404, 1987.

[60] J. R. Gilbert and S.-H. Teng. Matlab mesh partitioning and graph separator toolbox, 2002. http://www.cerfacs.fr/algor/Softs/MESHPART/index.html.

[61] R. L. Glass. The realities of software technology payoffs. *Communications of the ACM*, 42(2):74–79, 1999.

[62] M. T. Goodrich. Communication-efficient parallel sorting. In *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 247–256, 1996.

[63] K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report TR-2002-55, University of Texas at Austin, Department of Computer Science, Nov 2002.

[64] N. Goyal and E. Meiburg. Unstable density stratification of miscible fluids in a vertical hele-shaw cell: Influence of variable viscosity on the linear stability. *Journal of Fluid Mechanics*, 516:211–238, 2004.

[65] D. Gregor and A. Lumsdaine. The parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing (POOSC)*, July 2005.

[66] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software*, 4(3):250–269, 1978.

[67] K. Hall. An r-dimensional quadratic placement algorithm. *Management Science*, 17(3):219–229, Nov 1970.

[68] D. R. Helman, J. JáJá, and D. A. Bader. A new deterministic parallel sorting algorithm with an experimental evaluation. *J. Exp. Algorithmics*, 3, 1998.

[69] V. E. Henson and U. M. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41(1):155–177, 2002.

[70] L. Hochstein and V. R. Basili. An empirical study to compare two parallel programming models. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2006. ACM Press.

[71] R. Howard. *Dynamic probabilistic systems*. John Wiley, New York, 1971.

[72] P. Husbands. *Interactive supercomputing*. PhD thesis, Massachussetts Institute of Technology, 1999.

[73] P. Husbands, C. Isbell, and A. Edelman. MITMatlab: A tool for interactive supercomputing. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[74] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.

[75] Interactive Supercomputing LLC. *Star-P Software Development Kit (SDK): Tutorial and Reference Guide*, 2007. version 2.5.

[76] Interactive Supercomputing LLC. *Star-P User Guide*, 2007. version 2.5.

[77] E. W. Johnson and J. B. Brockman. Measurement and analysis of sequential design processes. *ACM Transactions on Design Automation of Electronic Systems*, 3(1):1–20, 1998.

[78] G. Kirchoff. Über den durchgang eines elektrischen stromees durch eine ebene, insbesondere durch eine kreisförmige. *Annalen der Physik und Chemie*, 64:497–514, 1845.

[79] D. J. Klein and M. Randic. Resistance distance. *Journal of Mathematical Chemistry*, 12(1):81–95, Dec 1993.

[80] N. B. Kotliar and J. A. Wiens. Multiple scales of patchiness and patch structure: A hierarchical framework for the study of heterogeneity. *Oikos*, 59(2):253–260, Nov 1990.

[81] S. Kullback and R. A. Leibler. On information and sufficiency. *Annals of Mathematical Statistics*, 22:79–86, 1951.

[82] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401:788–791, Oct 1999.

[83] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *Advances in Neural Information Processing Systems*, pages 556–562, 2000.

[84] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK Users Guide: Solution of Large Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, Philadelphia, 1998.

[85] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In A. Jorge, L. Torgo, P. Brazdil, R. Camacho, and J. Gama, editors, *PKDD*, volume 3721 of *Lecture Notes in Computer Science*, pages 133–145. Springer, 2005.

[86] X. S. Li and J. W. Demmel. SuperLU_DIST: A scalable distributed memory sparse direct solver for unsymmetric linear systems. *ACM Transactions on Mathematical Software*, 29(2):110–140, 2003.

[87] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1985. ACM Press.

[88] M. Mahr and M. Mauro. Making Science, making Change: Celebrating five years of research and collaboration in the Yellowstone to Yukon region, May 2003.

[89] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, Apr 1957.

[90] K. Maschhoff and D. Sorensen. A portable implementation of ARPACK for distributed memory parallel architectures. *Proceedings of Copper Mountain Conference on Iterative Methods*, Apr 1996.

[91] Mathworks Inc. *MATLAB User's Guide*, 2007. version 2007a.

[92] B. McRae. *Circuitscape User Manual*. Northern Arizona University, 2006. version 2.2.

[93] B. H. McRae. Isolation by resistance. *Evolution*, 60(8):1551–1561, 2006.

[94] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Geometric separators for finite-element meshes. *SIAM Journal on Scientific Computing*, 19(2):364–386, 1998.

[95] C. Min and F. Gibou. A second order accurate projection method for the incompressible navier-stokes equations on non-graded adaptive grids. *Journal of Computational Physics*, 219(2):912–929, 2006.

[96] J. Nesetril, E. Milkova, and H. Nesetrilova. Otakar Boruvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233:3–36, 2001.

[97] Netflix Inc. The Netflix Prize. http://www.netflixprize.com/.

[98] P. Plauger, M. Lee, D. Musser, A. A. Stepanov, and A. Stepanov. *C++ Standard Template Library*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[99] D. Post and R. Kendall. Software project management and quality engineering, practices for complex, coupled multi-physics, massively parallel

computational simulations: Lessons learned from ASCI. *Los Alamos National Laboratory*, 2003.

[100] A. Pothen, H. D. Simon, and K.-P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal of Matrix Analysis and Applications*, 11(3):430–452, 1990.

[101] S. Reinhardt, J. R. Gilbert, and V. Shah. High performance graph algorithms from parallel sparse matrices. In *Proceedings of the Workshop on state of the art in scientific and parallel computing*, 2006.

[102] A. Reiser. A linear selection algorithm for sets of elements with weights. *Information Processing Letters*, 7(3):159–162, 1978.

[103] C. Robertson. Sparse parallel matrix multiplication. Master's thesis, UC Santa Barbara, 2005.

[104] M. Roh and V. Shah. Investigation of amg as a preconditioner for quadtree discretizations of level set problems. http://gauss.cs.ucsb.edu/∼viral/amg/html/runall.html, Jun 2006.

[105] E. L. G. Saukas and S. W. Song. A note on parallel selection on coarse grained multicomputers. *Algorithmica*, 24(3/4):371–380, 1999.

[106] SGI. *The SGI Message Passing Toolkit*.

[107] J. N. Shadid and R. S. Tuminaro. Sparse iterative algorithm software for large-scale MIMD machines: An initial discussion and implementation. *Concurrency: Practice and Experience*, 4(6):481–497, 1992.

[108] V. Shah and J. R. Gilbert. Sparse matrices in Matlab*P: Design and implementation. In L. Bougé and V. K. Prasanna, editors, *HiPC*, volume 3296 of *Lecture Notes in Computer Science*, pages 144–155. Springer, 2004.

[109] H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.

[110] Y. Shiloach and U. Vishkin. An O(log n) parallel connectivity algorithm. *Journal of Algorithms*, 3(1):57–67, 1982.

[111] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost graph library: User guide and reference manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[112] M. Slatkin. Isolation by distance in equilibrium and non-equilibrium populations. *Evolution*, 47:264–279, 1993.

[113] B. Smith, J. Boyle, J. Dongarra, B. Garbow, Y. Ilebe, V. Kelma, and C. Moler. *Matrix Eigensystem Routines - EISPACK Guide*. Springer-Verlag, 2nd edition, 1976.

[114] B. Smith, D. Mizell, J. Gilbert, and V. Shah. Towards a timed Markov process model of software development. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 65–67, New York, NY, USA, 2005. ACM Press.

[115] Smith, Robert P. and Eppinger, Steven D. Identifying controlling features of engineering design iteration. *Management Science*, 43(3):276–293, Mar 1997.

[116] K. Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128:281–309, Mar 2001.

[117] R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, 1981.

[118] D. Urban and T. Keitt. Landscape connectivity: A graph-theoretic perspective. *Ecology*, 82(5):1205–1218, 2001.

[119] P. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. A talk based on the manuscript was presented at the IMA Workshop on Graph Theory and Sparse Matrix Computation, Oct 1991.

[120] H. A. van der Vorst. High performance preconditioning. *SIAM J. Sci. Stat. Comput.*, 10(6):1174–1185, 1989.

[121] G. van Rossum. *Python Reference Manual*. Python Software Foundation, 2006. version 2.5.

[122] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics Conference Series*, 16:521–530, Jan. 2005.

[123] C. Walston and C. Felix. A method of programming measurement and estimation. *IBM Journal of Research and Development*, 1977.

[124] S. Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[125] R. C. Waters. The Series macro package. *SIGPLAN Lisp Pointers*, III(1):7–11, 1990.

[126] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[127] M. Zelkowitz, V. Basili, S. Asgari, L. Hochstein, J. Hollingsworth, and T. Nakamura. Measuring productivity on high performance computers. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, Washington, DC, USA, 2005. IEEE Computer Society.