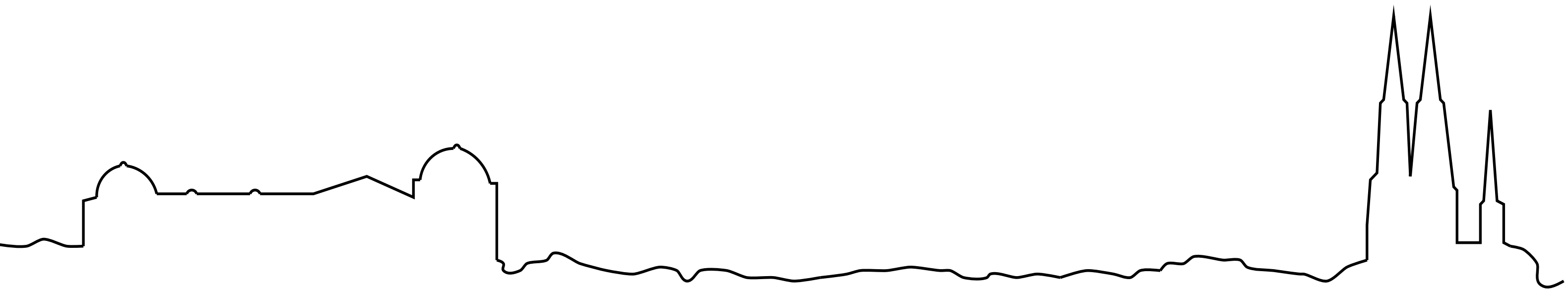
Investigating Google's PageRank algorithm

by

Erik Andersson
eran3133@student.uu.se

Per-Anders Ekström
peek5081@student.uu.se

Report in Scientific Computing, advanced course - Spring 2004



UPPSALA UNIVERSITET
Inst. för informationsteknologi
Avd. för teknisk databehandling

UPPSALA UNIVERSITY
Information Technology
Dept. of Scientific Computing

Abstract

This paper presents different parallel implementations of Google's PageRank algorithm. The purpose is to compare different methods for computing PageRank on large domains of the Web. The iterative algorithms used are the Power method and the Arnoldi method.

We have implemented these algorithms in a parallel environment and created a basic Web-crawler to gather test data. Tests have then been carried out with the different algorithms using various test data.

The explicitly restarted Arnoldi method was shown to be superior to the normal Arnoldi method as well as the Power method for high values of the dampening factor α. Results also show that load balancing our parallel implementation was usually quite ineffective.

For smaller values of α, including 0.85 as Google uses, the Power method is preferable. It is usually somewhat slower, but the memory used is significantly less. For higher values of α, if very accurate results are needed, the restarted Arnoldi method is preferable.

Contents

1 Introduction

Search engines are huge power factors on the Web, guiding people to information and services. Google[1] is the most successfull search engine in recent years, mostly due to its very comprehensive and accurate search results. When Google was an early research project at Stanford, several papers were written describing the underlying algorithms [2] [3]. The dominant algorithm was called PageRank and is still the key for providing accurate rankings for search results.

Google uses the Power method to compute PageRank. For the whole Internet and larger domains this is probably the only possible method (principally due to the high memory-requirements of other methods). In the Power method a number (50-100) of matrix vector multiplications are performed.

For smaller domains, other methods than the Power method would be interesting to investigate. One good candidate is the Arnoldi method which has higher memory requirements but converges after less iterations.

To efficiently handle these large-scale computations we need to implement the algorithms using a parallel system. Some sort of load balancing might be needed to get good performance for the parallelization.

A Web-crawler needs to be implemented to gather realistic test data.

In this review we investigate these methods.

2 Review of PageRank

In this following section we present the basic ideas of PageRank. We also describe various problems for calculating PageRank and how to resolve them.

2.1 PageRank explained

The Internet can be seen as a large graph, where the Web pages themselves represent nodes, and their links (direct connection to other Web pages) can be seen as the edges of the graph. The links (edges) are directed; i.e. a link only points one way, although there is nothing stopping the other page from pointing back. This interpretation of the Web opens many doors when it comes to creating algorithms for deciphering and ranking the world's Web-pages.

The PageRank algorithm is at the heart of the Google search engine. It is this algorithm that in essence decides how important a specific page is and therefore how high it will show up in a search result.

The underlying idea for the PageRank algorithm is the following: *a page is important, if other important pages link to it.* This idea can be seen as a way of calculating the importance of pages by voting for them. Each link is viewed as a vote - a de facto

[1]http://www.google.com

recommendation for the importance of a page - whatever reasons the page has for linking to a specific page. The PageRank-algorithm can, with this interpretation, be seen as the counter of an online ballot, where pages vote for the importance of others, and this result is then tallied by PageRank and is reflected in the search results.

However, not all votes are equally important. A vote from a page with low importance (i.e. it has few *inlinks*[2]) should be worth far less than a vote from an important page (with thousands of inlinks). Also, each vote's importance is divided by the number of different votes a page casts, i.e. with a single *outlink*[3] all the weight is put towards the sole linked page, but if 100 outlinks are present, they all get a 1/100th of the total weight.

For n pages $P_i, i = 1, 2, \ldots, n$ the corresponding PageRank is set to $r_i, i = 1, 2 \ldots, n$. The mathematical formulation for the recursively defined PageRank are presented in equation (1):

$$r_i = \sum_{j \in L_i} r_j / N_j, \qquad i = 1, 2, \ldots, n. \tag{1}$$

where r_i is the PageRank of page P_i, N_j is the number of outlinks from page P_j and L_i are the pages that link to page P_i.

Since this is a recursive formula an implementation needs to be iterative and will require several iterations before stabilizing to an acceptable solution. Equation (1) can be solved in an iterative fashion using algorithm (2.1):

Algorithm 2.1 PageRank

```
1: $r_i^{(0)}$,   $i = 1, 2 \ldots, n.$   arbitrary nonzero starting value
2: for $k = 0, 1, \ldots$ do
3:   $r_i^{(k+1)} = \sum_{j \in L_i} \frac{r_j^{(k)}}{N_j}$,   $i = 1, 2 \ldots, n.$
4:   if $\|\mathbf{r}^{(k)} - \mathbf{r}^{(k+1)}\|_1 < tolerance$ then
5:     break
6:   end if
7: end for
```

You start with an arbitrarily guessed vector r (e.g. a vector of ones, all divided with number of pages present), that describes the initial PageRank value r_i for all pages P_i. Then you iterate the recursive formula until two consecutively iterated PageRank vectors are similar enough.

[2] An inlink is a link that points to the current page from another page.
[3] An outlink is a link that points out from the current page to another page.

2.2 Matrix model

By defining a matrix

$$Q_{ij} := \begin{cases} 1/N_i & \text{if } P_i \text{ links to } P_j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

the PageRank problem can be seen as a matrix-problem. The directed graph in Figure 1 exemplifies a very small isolated part of the Web with only 6 Web-pages, $P_1, P_2, \ldots, P_6$.

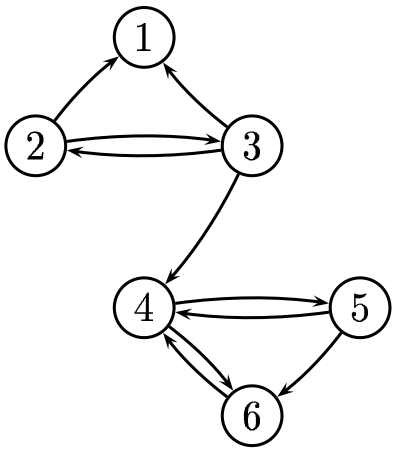


Figure 1: Small isolated Web site of 6 pages $P_1, P_2, \ldots, P_6$

In the matrix-formulation, this link structure will be written as:

$$\mathbf{Q} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \tag{3}$$

Here Q_{ij} describes that there is a link from page P_i to page P_j, and these are all divided by N_i (which is the number of outlinks on page P_i).

The iteratively calculated PageRank $\mathbf{r}$ could then be written as:

$$\mathbf{r}^T_{(k+1)} = \mathbf{r}^T_{(k)}\mathbf{Q} \qquad , k = 0, 1, \ldots \tag{4}$$

i.e. the Power method.

2.3 Random walker

To better explain and visualize the problems and concepts of the PageRank-algorithm in an intuitive fashion, a random walker model of the web can be used. This random walker (or surfer) starts from a random page, and then selects one of the outlinks from the page in a random fashion. The PageRank (importance) of a specific page can now be viewed as the asymptotic probability that the surfer is present at the page. This is possible, as the surfer is more likely to randomly wander to pages with many votes (lots of inlinks), giving him a large probabibility of ending up in such pages.

2.3.1 Stuck at a page

The random walker described above will run into difficulties on his trek around the web. As he randomly "wanders" through the link structure he might reach a page that has no outlinks - forever confining him to this page. For the small Web shown in figure 1 this will happen if the random walker goes to page P_1. The corresponding link-matrix has a row of zeros at every page without outlinks. How can this problem be solved?

The method used in the PageRank-algorithm is the following:
Replace all zeros with $1/n$ in all the zero-rows, where n is the dimension of the matrix.

In our matrix formulation, this can be written as:

$$\hat{\mathbf{Q}} = \mathbf{Q} + \frac{1}{n}\mathbf{d}\mathbf{e}^T \tag{5}$$

where $\mathbf{e}$ is a column-vector of ones, and $\mathbf{d}$ is a column-vector that describe which rows in the matrix $\mathbf{Q}$ that are all zero, it's defined as

$$d_i := \begin{cases} 1 & \text{if } N_i = 0 \\ 0 & \text{otherwise} \end{cases}, i = 1, 2, \ldots, n. \tag{6}$$

For our example matrix this addition would be:

$$\hat{\mathbf{Q}} = \mathbf{Q} + \frac{1}{n}\mathbf{d}\mathbf{e}^T = \mathbf{Q} + \frac{1}{6}\begin{pmatrix}1\\0\\0\\0\\0\\0\end{pmatrix}\begin{pmatrix}1 & 1 & 1 & 1 & 1 & 1\end{pmatrix} =$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} + \begin{pmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

With the creation of matrix $\hat{\mathbf{Q}}$, we have a row-stochastic matrix, i.e. a matrix where all rows sums up to 1.

2.3.2 Stuck in a subgraph

There is still one possible pitfall for our random walker, he can wander into a subsection of the complete graph that does not link to any outside pages, locking him into a small part of the web. For the small Web shown in Figure 1 this will happen if the walker comes down to the lower part of the structure. If he ends up in this section, there are no possibilities for him to return to the upper part. In the link-matrix described above this corresponds to an reducible matrix.

This means that if he gets to the enclosed subsection he will randomly wander inside this specific subgraph, and the asymptotic probability that he will be in one of these pages will increase with each random step. We therefore want the matrix to be irreducible, making sure he can not get stuck in a subgraph.

The method used in PageRank to guarantee irreducibility is something called "teleportation", the ability to jump, with a small probability, from any page in the linkstructure to any other page. This can mathematically be described as:

$$\hat{\hat{\mathbf{Q}}} = \alpha\hat{\mathbf{Q}} + (1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T \tag{7}$$

where $\mathbf{e}$ is a column-vector of ones, and α is a dampening factor (i.e. the "teleportation" probability factor). For our example matrix and an α set to 0.85 this addition would be:

$$\begin{aligned}
\hat{\hat{\mathbf{Q}}} &= \alpha\hat{\mathbf{Q}} + (1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T = \\
&= 0.85\begin{pmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} + (1-0.85)\frac{1}{6}\begin{pmatrix}1\\1\\1\\1\\1\\1\end{pmatrix}\begin{pmatrix}1 & 1 & 1 & 1 & 1 & 1\end{pmatrix} = \\
&= \frac{17}{20}\begin{pmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & \frac{1}{3} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} + \frac{1}{40}\begin{pmatrix} 1&1&1&1&1&1\\1&1&1&1&1&1\\1&1&1&1&1&1\\1&1&1&1&1&1\\1&1&1&1&1&1\\1&1&1&1&1&1 \end{pmatrix} = \\
&= \begin{pmatrix} 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \\ 11/12 & 1/60 & 11/12 & 1/60 & 1/60 & 1/60 \\ 19/60 & 19/60 & 1/60 & 19/60 & 1/60 & 1/60 \\ 1/60 & 1/60 & 1/60 & 1/60 & 7/15 & 7/15 \\ 1/60 & 1/60 & 1/60 & 7/15 & 7/15 & 1/60 \\ 1/60 & 1/60 & 1/60 & 11/12 & 1/60 & 1/60 \end{pmatrix}
\end{aligned}$$

With the creation of matrix $\hat{\hat{\mathbf{Q}}}$, we have an *irreducible matrix*[4].

[4]http://mathworld.wolfram.com/IrreducibleMatrix.html

When adding $(1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T$ there is an equal chance of jumping to all pages. Instead of $\mathbf{e}^T$ we can use a weighted vector, having different probabilities for certain pages - this give us power to bias the end-result for our own needs.

2.4 The selection of α

It may be shown [4] that if our matrix $\hat{\mathbf{Q}}^T$ has eigenvalues: $\{1, \lambda_2, \lambda_3, \ldots\}$ our new matrix $\hat{\hat{\mathbf{Q}}}^T$ will have the eigenvalues: $\{1, \alpha\lambda_2, \alpha\lambda_3, \ldots\}$.

The value α therefore heavily influences our calculations. It can be shown that the Power method approximately converges at the rate of $C \mid \lambda_2/\lambda_1 \mid^{(m)}$, and as we created our final matrix $\hat{\mathbf{Q}}^T$, we scaled down all eigenvalues (as shown above) except the largest one with the factor α.

- A small α (≈ 0.7) would therefore mean that the Power method converges quickly. This also means that our final result would poorly describe the underlying link structure as we allow the teleportation to heavily influence the result.
- A large α (≈ 0.9) on the other hand means that the Power method converges slowly, but the answer better describes the properties of the real underlying link structure.

As a good compromise, Google uses an α of 0.85 [2].

2.5 Practical calculations of PageRank

$\hat{\hat{\mathbf{Q}}}$ is an irreducible row-stochastic matrix. According to Perron-Frobenius Theory [4], an irreducible column-stochastic matrix has 1 as the largest eigenvalue and its corresponding right eigenvector has only non-negative elements. That is the reason we do a left hand multiplication. We now have everything we need to compute PageRank, and the final formula becomes:

$$\hat{\hat{\mathbf{Q}}}^T\mathbf{r} = \mathbf{r} \tag{8}$$

The form of the problem written in equation (8) is the classic definition of the eigenvalue/vector-problem, and the goal of finding the importance of all pages transforms into the problem of finding the eigenvector corresponding to the largest eigenvalue of 1.

As written above, the matrices used in the PageRank-calculations are immense, but it can be shown that we do not have to create the full matrix $\hat{\hat{\mathbf{Q}}}^T$, nor the somewhat full matrix $\hat{\mathbf{Q}}^T$ explicitly to correctly calculate PageRank. We can instead directly use our very sparse link matrix $\mathbf{Q}$, which was initially created to describe the link structure together with two more sparse matrices, as in equation (9).

$$\mathbf{r} = \hat{\hat{\mathbf{Q}}}^T\mathbf{r} = \alpha\hat{\mathbf{Q}}^T\mathbf{r} + (1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T\mathbf{r} = \alpha\mathbf{Q}^T\mathbf{r} + \alpha\frac{1}{n}\mathbf{e}\mathbf{d}^T\mathbf{r} + (1-\alpha)\frac{1}{n}\mathbf{e}\mathbf{e}^T\mathbf{r} \tag{9}$$

3 Eigenvector computing

Since PageRank is the same as the eigenvector corresponding to the largest (right) eigenvalue of the matrix $\hat{\hat{\mathbf{Q}}}^T$ we need an iterative method that works well for large sparse matrices.

The Power method is an old and in many cases obsolete method. However, since it only needs one vector except for the unmodified matrix it does have some practical value for these kinds of calculations with high memory requirements, which is why it is used by Google. There are however better methods that can be used when we are only interested in a subset of the entire internet, say a university's link structure or a small country's. With these smaller matrices we can use more memory for our calculations. The Arnoldi-method (see section 3.2) seems very well suited for these instances.

3.1 Power Method

The Power method is a simple method for finding the largest eigenvalue and corresponding eigenvector of a matrix. It can be used when there is a dominant eigenvalue of A. That is, the eigenvalues can be ordered such that $\lambda_1 > \lambda_2 \geq \lambda_3 \geq \ldots \geq \lambda_n$. λ_1 must be strictly larger than λ_2 that is in turn larger than or equal to the rest of the eigenvalues.

The basic algorithmic version of the Power method can be written as:

Algorithm 3.1 Normalized Power Method

```
1: $\mathbf{x}_0$ = arbitrary nonzero starting vector
2: for $k = 1, 2, \ldots$ do
3:     $\mathbf{y}_k = \mathbf{A}\mathbf{x}_{k-1}$
4:     $\mathbf{x}_k = \mathbf{y}_k / \|\mathbf{y}_k\|_1$
5: end for
```

This algorithm fails if the chosen starting vector is perpendicular to the true eigenvector.

The rate of convergence may be shown to be linear for the Power method, thus $|\lambda_1 - \lambda_1^{(m)}| \approx C|\lambda_2/\lambda_1|^{(m)}$, where C is some positive constant. This is of great interest for our PageRank calculations since we can influence the size of λ_2 by changing α.

3.1.1 Using the Power Method for PageRank

Because the matrix used in PageRank will be very large (each row represents a page and each entry represents a link) very few methods can successfully be used to calculate the PageRank. The Power method described above is the one utilized by Google as it has some very redeeming qualities:

- We only need to save the previous approximated eigenvector.
- It finds the eigenvalue and vector for the largest eigenvalue, which is what we are interested in.
- It does not in any way alter our matrix.

Since the vector x in algorithm 3.1 has the norm $\|\mathbf{x}\|_1 = \mathbf{e}^T\mathbf{x} = 1$, then $\|\mathbf{y}\|_1 = \mathbf{e}^T\mathbf{y} = \mathbf{e}^T\mathbf{A}\mathbf{x} = \mathbf{e}^T\mathbf{x} = 1$ since A is column-stochastic ($\mathbf{e}^T\mathbf{A} = \mathbf{e}^T$). Therefore the normalization step in algorithm 3.1 is unnecessary.

By using the Power method to calculate PageRank it can also be shown [4] that $\mathbf{r}$ can be calculated by

$$\mathbf{r} = \alpha\mathbf{Q}^T\mathbf{r} + \frac{1}{n}\mathbf{e} - \|\alpha\mathbf{Q}^T\mathbf{r}\|_1$$

instead of as in equation (9). So we do not need to know $\mathbf{d}$, i.e. we do not need to know which pages that lack outlinks.

3.2 Arnoldi Method

The Arnoldi method is a *Krylov subspace*[5] method that can be used to iteratively find all eigenvalues and their corresponding eigenvectors for a matrix A. It was first created and used for transforming a matrix into *upper Hessenberg form*[6] [1], but it was later seen that this method could successfully be used to find eigenvalues and eigenvectors for a large sparse matrix in an iterative fashion. The method starts by building up bases for the Krylov-subspace:

Algorithm 3.2 Arnoldi Method

```
1:  $\mathbf{v}_0$ = arbitrary nonzero starting vector
2:  $\mathbf{v}_1 = \mathbf{v}_0/\|\mathbf{v}_0\|_2$
3:  for $j = 1, 2, \dots$ do
4:      $\mathbf{w} := \mathbf{A}\mathbf{v}_j$
5:      for $i = 1 : j$ do
6:          $h_{ij} = \mathbf{w}^*\mathbf{v}_i$
7:          $\mathbf{w} := \mathbf{w} - h_{ij}\mathbf{v}_i$
8:      end for
9:      $h_{j+1,j} = \|\mathbf{w}\|_2$
10:     if $h_{j+1,j} = 0$ then
11:         stop
12:     end if
13:     $\mathbf{v}_{j+1} = \mathbf{w}/h_{j+1,j}$
14: end for
```

After we have created the subspace, with m as a chosen amount of bases, we can calculate approximations of the eigenvalues and eigenvectors of the original sparse matrix $\mathbf{A}$.

[5] A Krylov subspace is defined as $K(A, q, j) = span(q, Aq, A^2q, \dots A^{j-1}q)$.

[6] An upper Hessenberg matrix has zero entries below the first subdiagonal.

The $m \times m$ Hessenberg matrix $\mathbf{H}$ created above is the key here - the eigenvalues associated with it, $\lambda_i^{(m)}$ are known as *Ritz Values* and will converge, with more and more bases for the Krylov subspace, towards the eigenvalues of the large sparse matrix $\mathbf{A}$.

The eigenvectors for the matrix $\mathbf{A}$ can then be calculated as follows:

- Retrieve a specific eigenvalue of $\mathbf{H}$ - that we are interested in.
- Retrieve the eigenvector of $\mathbf{H}$ associated with this value.
- The corresponding eigenvector for $\mathbf{A}$ can then be found by

  $$\mathbf{u}_i^{(m)} = V_m y_i^{(m)} \tag{10}$$

  where $\mathbf{u}_i^{(m)}$ is the corresponding eigenvector in $\mathbf{A}$, $\mathbf{V}_m$ is the vector with bases for the Krylov subspace and $\mathbf{y}_i^{(m)}$ is the eigenvector from $\mathbf{H}$ associated with a specific eigenvalue.

3.2.1 Stopping criterion

The residual norm of the PageRank vector is used to decide when to stop the iterations. When the residual between two consecutive iterations changes less than a certain tolerance we stop iterating.

In the Arnoldi method we can use a very computationally cheap method to find out a stopping criterion instead of directly calculating the residual of two consecutive (approximated) eigenvectors. We do this by obtaining the residual norm for the *Ritz pair*[7]. This method is very inexpensive [1] and is therefore one of the advantages of the Arnoldi method. The cheap way of computing the norm is described in equation (11).

$$\|(\mathbf{A} - \lambda_\mathbf{i}^{(\mathbf{m})}\mathbf{I})\mathbf{u}_i^{(m)}\|_2 = h_{m+1,m}|\mathbf{e}_m^*\mathbf{y}_i^{(m)}| \tag{11}$$

3.2.2 Using the Arnoldi Method for PageRank

The iterative algorithm for finding a specific eigenvalue and eigenvector becomes:

Initial:

- Create an initial basis, usually uniform.

For $m = 1, 2, \ldots$

- Add an extra basis to your subspace.
- Calculate the eigenvector/eigenvalue we are interested in from the Hessenberg-matrix.
- **If** $h_{m+1,m}|\mathbf{e}_m^*\mathbf{y}_i^{(m)}| <$ tol **Break**.

Final:

- Find the corresponding eigenvector in the real matrix as in equation (10).

[7]The Ritz pair is the approximate eigenpair $(\mathbf{y}_i^{(m)}, \lambda_i^{(m)})$.

3.2.3 Explicit restart

When the number of iterations grows, the amount of work required for the Arnoldi method increases rapidly. Additional work, except for the matrix vector multiplication used in the Power method, is the work to orthogonalize the newly iterated Arnoldi basis against all of the previous ones. The Ritz values and vectors, i.e. the eigenvalues and eigenvectors of the Hessenberg matrix, also need to be computed after each iteration. All this extra work increases by each iteration as there will be more vectors to orthogonalize against and the Hessenberg matrix will grow.

The idea of explicit restart is to perform m number of steps in algorithm 3.2, compute the approximate eigenvector $\mathbf{u}_i^{(m)}$, i.e. the PageRank vector, end if satisfied with the results, else restart the algorithm 3.2 with initial vector $\mathbf{v}_0 = \mathbf{u}_i^{(m)}$.

3.3 Accuracy of the eigenvector

The resulting eigenvector that describes the importance of all the pages in our link structure is a probability vector; all elements are between 0 and 1, and the vector sums up to 1. This means that for very large partitions of the web (ranging from millions to billions of Web pages) each individual entry in this vector will be very small. Out of this comes an intrinsic demand for a very accurate representation of the numbers in the eigenvector, so that we can correctly determine the relative importance of pages. It can be shown that for any real ranking of the web, with the number of pages in the range of billions, we need an accuracy in the order of at least 10^{-9}. But as pages belonging to the same query usually do not have very similar PageRank-values an accuracy of 10^{-12} is probably the most accurate accuracy that will be needed [6].

4 Sparse Matrix formats

As the dimension of the link matrix grows, its relative "sparseness" increases aswell. To compute PageRank for large domains there are no possible way to work with the matrix in its full format, the memory requirements would be too high. Therefore we use sparse matrix formats.

4.1 Compress Row Storage (CRS)

We have chosen to store our sparse matrices row-oriented, i.e. the matrix is represented by a sequence of rows.

The Compress Row Storage format is one of the most extensively used storage scheme for general sparse matrices, with minimal storage requirements. Algorithms for many important mathematical operations are easily implemented, for example matrix vector multiplication (SpMxV). A problem using SpMxV with this sparse-format though is the extremely bad data locality for the vector we are multiplying with - as we randomly jump to elements in it.

In Table 1 we illustrate the CRS storage scheme of matrix **Q** (3).

index	1	2	3	4	5	6	7	8	9	10
row_ptr	1	1	3	6	8	10	11			
col_ind	1	3	1	2	4	5	6	4	5	4
val	1/2	1/2	1/3	1/3	1/3	1/2	1/2	1/2	1/2	1

Table 1: CRS storage scheme for matrix **Q** (3)

Using the CRS scheme we claimed that algorithms for many mathematical operations were very simple to implement. To illustrate this, we present the most important operation in this report: SpMxV, i.e. sparse matrix vector multiplication.

Algorithm 4.1 SpMxV

```
1: for $i = 0 : dim$ do
2:   for $j = \textbf{row\_ptr}[i] : \textbf{row\_ptr}[i+1]$ do
3:     $\textbf{sol}[i] = \textbf{sol}[i] + \textbf{val}[j] \cdot \textbf{v}[\textbf{col\_ind}[j]]$
4:   end for
5: end for
```

Transposed sparse matrix vector multiplication is as easy to implement. There will only be a small difference in row (3) [7].

The problem with these implementations is, as mentioned above, the terrible data locality. Index values from **col_ind** will make random jumps in the vector **v**. Thus there will be cache misses in vector **v** for almost every iteration.

4.2 MATLAB internal sparse format.

MATLAB uses its own simple storage scheme for sparse matrices. For each non zero element in a sparse matrix, MATLAB stores a (x,y,val) triple to describe the position and the value of this element.

In Table 2 we illustrate the MATLAB storage scheme of matrix **Q** (3).

index	1	2	3	4	5	6	7	8	9	10
(x,y)	(1,2)	(3,2)	(1,3)	(2,3)	(4,3)	(5,4)	(6,4)	(4,5)	(5,5)	(4,6)
val	1/2	1/2	1/3	1/3	1/3	1/2	1/2	1/2	1/2	1

Table 2: MATLAB storage scheme for matrix **Q** (3)

5 Parallel implementation

Implementing the PageRank-calculations in a parallel environment opens several possibilities in partitioning the data (i.e. how the data is divided by the processors) and load balancing the data (i.e. to ensure that all processors do the same amount of work). When we try to load balance and partition the data there are several issues that must be weighted together, for example a good partitioning for one specific operation might give us problems for others.

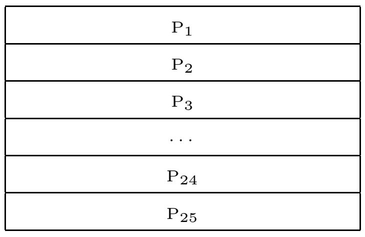
5.1 Partitioning

The most expensive operations done in the calculation of the PageRank-values are matrix-vector multiplications, and it is a perfectly parallel operation with several possible methods of partitioning both the matrix and the vector.

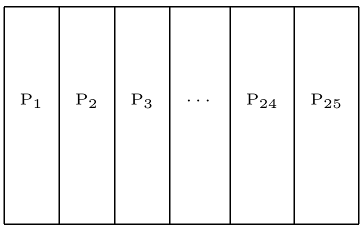
We have considered three different methods for partitioning the link matrix among the processors.

- Divide the matrix using a row-wise distribution.
- Divide the matrix using a column-wise distribution.
- Divide the matrix as a 2D cartesian grid.

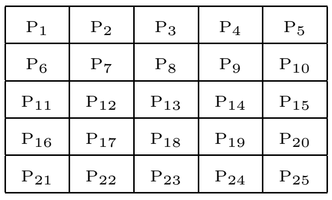
Figure 2 visualizes the three different schemes.



(a) Row-wise partitioning



(b) Column-wise partitioning


P_1 P_2 P_3 P_4 P_5
P_6 P_7 P_8 P_9 P_{10}
P_{11} P_{12} P_{13} P_{14} P_{15}
P_{16} P_{17} P_{18} P_{19} P_{20}
P_{21} P_{22} P_{23} P_{24} P_{25}

(c) Partitioning using a 2D cartesian grid

Figure 2: Matrix partitioned on 25 processors using three different partitioning schemes

The method chosen for our computations is the row-wise partitioning shown in Figure 2(a). The reason for using this method is that the matrix itself is stored in a (sparse) row-wise format (see section 4), and any efficient partitioning must utilize the underlying storage-structure.

This also means that all processors have their own small part of the vector that is calculated in this matrix-vector multiplication. All these parts must then be gathered together by all processor to build the complete vector that was calculated in this multiplication.

The vector used in the multiplication can also be divided among the processors, in several ways:

- Don't divide the vector at all, each processor holds a full copy of the vector we are multiplying with. Costs memory but saves in communication.
- Divide the vector into parts to go with the row-wise partitioning described above. This means that all processor hold a small part of the vector we are multiplying with. They multiply with all elements in their part of the matrix that they can. The processor then sends its part to the processor above, and receive from the processor below. This saves on memory but demands more communication.

The method used when we calculate PageRank is the first one, as our problems are quite small in comparison, giving us a final partitioning as follows:

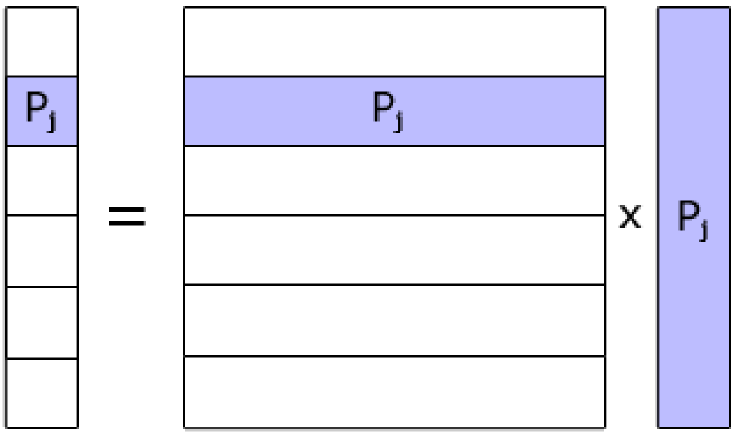


Figure 3: The parts of the matrices processor P_j stores

This gives us an iterative method for doing consecutive matrix$\times$vector-multiplications:

Startup:

- Distribute the rows of the Matrix in some fashion.
- All processors calculate the initial vector to multiply with, usually 1/n (where n is size of matrix).

Loop:

- All processors calculate their part of the result by multiplying their part of the matrix with the full vector that they have.
- All processor gather the new resulting vector and use it as the vector to multiply with in the next iteration.

We must also parallelize residual and norm-calculations, the most basic method, and the one used, is for each processor to calculate the norm/residual of their part of the result-vector - and then all processors sum up all the residuals/norms found in each processor.

5.2 Load Balancing

The initial idea of partitioning, where each processor gets the same amount of rows (as long as possible) is naive. In the link matrix used to calculate PageRank the number of non zero elements per row can differ immensely. This lends credence to the idea that the way to balance the calculations is by dividing the matrix so that each processor has to handle the same amount of non zero elements.

To enable this load balancing in the light of the storage format, we use a very simple method:

1. Each processor reads the file and receives the number of non zero elements and the number of rows.
2. All processors read each row-pointer, but the processor who doesn't want the specific row-info just throws away the data.
3. When a processor has retrieved so many rows so that it meets the calculated amount of non zero elements it should use, it just start throwing away data.
4. After each processor has read the rows they want, the specific info is read from col_ind and val-vectors in the file, all other info is thrown away.

5.2.1 Issues with Load Balancing

There are problems with this type of load balancing. As each processor read enough rows to meet their demands for non-zero elements or more, the final processor will end up with too few rows to read, giving it too few non zero elements in his part of the matrix. This factor is usually negligible for larger matrices.

Another issue with this load balancing stems from the Arnoldi-method. In this method (see section 3.2) the matrix$\times$vector isn't as large a part of the problem as in the Power method, but the load balancing described above was designed to minimize the problems of calculating this operation. The problem with Arnoldi is that we also normalize the new basis against all others and do several vector$\times$vector calculations. This demands that we also balance the size of each element's piece of the vector calculated in the matrix$\times$vector-multiplication. But with the method chosen, the subvectors resulting from the calculations and normalized in each processor will have the same amount of elements as the number of rows each processor have in the link-matrix. So if we correctly balance the matrix, giving the processors large differences in the number of rows, but the same amount of non zero elements, each processor will do widely amount of work when we also need to do much work with the sub-vector calculated in the matrix$\times$vector-multiplication.

6 Implementing a simple Web-Crawler

A Web-crawler[8] is a program that autonomously traverses ("crawls") the hyperlink structure building up the Web. It starts at a given Web-page (or a set of Web-pages), parses through their text looking for outgoing links, downloads the referred pages and so on recursively, until there are no more unvisited pages to be found, or some specified conditions are fulfilled.

To build up the link structure of a certain domain we only need to construct a very simple crawler. It needs to traverse all links recursively at the specified domain and save all the outlinks at every page.

Our implementation of a Web-crawler is built upon the skeleton described in hack #46 [5]. The page visiting order is breadth-first. Breadth-first is the most common way for crawlers to follow links. The idea is to retrieve all pages around the starting point before crawling deeper down, using a first in first out (FIFO) queuing system. Doing this we distribute the work load for the hosting servers, not hammering one single server at a time. With a breadth-first order we could also do a distributed (parallel) implementation of the crawler more easily than if we visited the pages in a depth-first order. In the other way to follow links, depth-first, we would follow the first link on the first page, then the first link on the second page and so on, until we meet a bottom, then the next link at the first page and so on.

The depth to crawl is not fixed since we want to crawl every single page of the specified domain, therefore it runs until there are no more new pages to visit.

[8] Also called: robot, spider, bot, worm, wanderer, gatherer, harvester...

A real commercial crawler would use multiple connections to different pages at the same time to remove the major bottleneck, the downloading time of each page. Google uses fast distributed crawlers with many multiple connections open at once [3]. Our crawler process every page serially, and is therefore very slow, especially on domains that are far away (ping-wise).

Our implementation traverses a full domain and saves the link structure in hash-tables of arrays. Algorithm 6.1 shows how the hash-tables are being filled during a crawl. Afterwards it's easy to loop through the hashes and save the matrices to file in chosen format.

Algorithm 6.1 Simple Web-Crawler to save link structure

```
1:  push(todo_list,initial_set_of_urls)
2:  while todo_list[0] $\neq \emptyset$ do
3:    page $\leftarrow$ fetch_page(todo_list[0])
4:    if page downloaded then
5:      links $\leftarrow$ parse(page)
6:      for all $l$ in links do
7:        if $l$ in done_list then
8:          push(todo_list[0].outlinks,done_list[$l$].id)
9:        else if $l$ in todo_list then
10:         push(todo_list[0].outlinks,todo_list[$l$].id)
11:       else if $l$ pass our filter then
12:         push(todo_list,$l$)
13:         todo_list[$l$].id = no. of url's
14:         push(todo_list[0].outlinks,todo_list[$l$].id)
15:       end if
16:     end for
17:   end if
18: end while
```

Saving the link structure by pushing the inlinks to every page (instead of the outlinks) we can also create the transposed matrix directly.

Our implemented Web-crawler can be seen in appendix A.3. It saves the matrix in either CRS (section 4.1) or MATLAB sparse format (section 4.2).

7 Test data

To test our implementations we have used three matrices. They are very different both when it comes to size as well as structure. One has been generated in MATLAB, one has been retrieved by crawling, and one was downloaded from the Web.

7.1 Stanford Web Matrix

This matrix was found at a research page at Stanford University[9]. It describes the link structure of the stanford.edu-domain from a September 2002 crawl and contains 281903 pages with about 2.3 million links. Stored in MATLAB sparse format it takes up 64.2MB, in CRS format it takes up about the same space. The sparsity pattern of the upper left part of the Stanford Web Matrix is visualized in Figure 4.

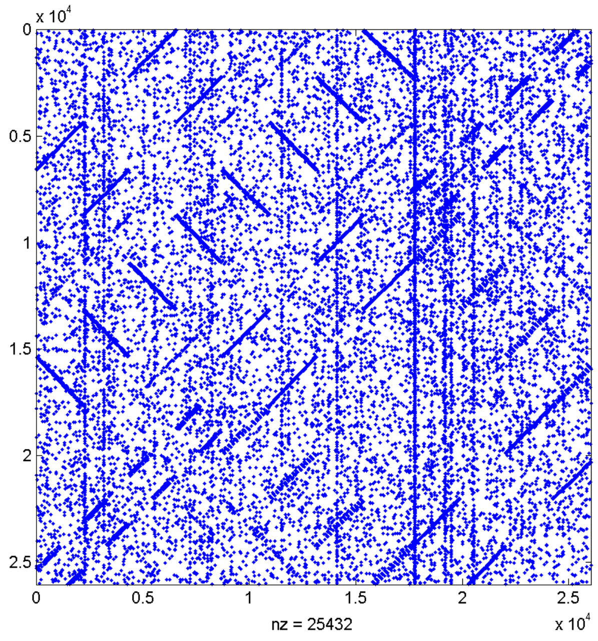


Figure 4: Upper left corner of Stanford Web Matrix

7.2 Crawled Matrix

We have used our implemented Web-crawler (see section 6) to obtain the link structure of a domain of our choice. Since the crawler is implemented in a serial fashion, it waits until a page is downloaded until it gets to download the next in line, it becomes very slow if there isn't a very good connection between the crawler and the web servers. Therefore we had to restrict our choice of domain to crawl, since running it on a large domain outside

[9]http://www.stanford.edu/ sdkamvar/research.html

our intranet would be quite unfeasible. We tried to get the whole uu.se domain, but after three days we realized that it would be to slow to finish in the nearby future.

The choice of domain to crawl thereafter came quite naturally since our own it-department probably has the largest sub-domain of Uppsala University. Crawling the whole it.uu.se-domain took a couple of hours on a good day. The last crawl we did was in april 2004 and it contains 46058 pages and 687635 links and takes up 11.2MB in the CRS format. The full sparsity pattern of our crawled matrix is visualized in Figure 5.

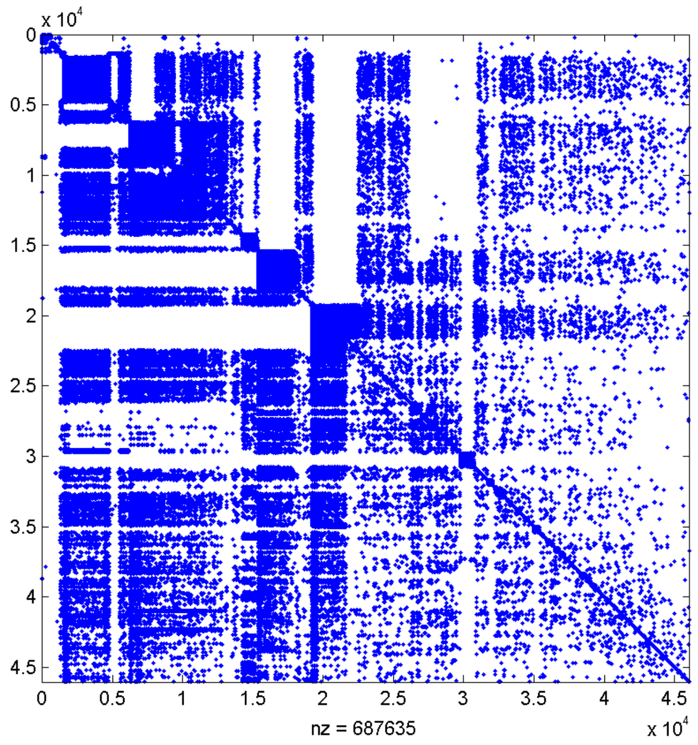


Figure 5: it.uu.se matrix

7.3 Randomly generated Column Stochastic Matrix

Using the following code we can generate a column stochastic matrix in Matlab. Input parameter is the dimension of the matrix we want. The generated matrix will get between 0 and 15 links on each row.

Algorithm 7.1 MATLAB code

```
function A = createMatrix(dim)
  A = sparse(dim,dim);
  maxnel = min(16,dim);

  for i = 1:dim
    nel = floor(rand(1)*maxnel);
    if(nel == 0)
      val = 0;
    else
      val = 1/nel;
    end
    for j = 1:nel
      col_ind = ceil(rand(1)*dim);
      while(A(col_ind,i) ~= 0)
        col_ind = ceil(rand(1)*dim);
      end
      A(col_ind,i) = val;
    end
  end
```

The matrix we generated and did tests on simulates a link structure of 1000.000 urls with an average of 7.56 links/url. Uncompressed in CRS format it takes up a space of about 135 MB. The sparsity pattern of the upper left part of this randomly generated matrix is visualized in Figure 6.

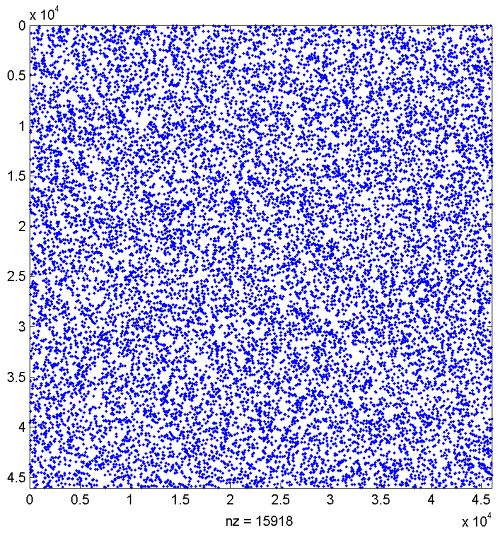


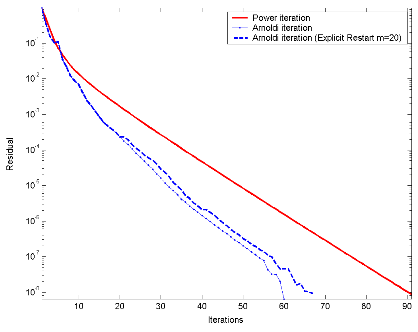
Figure 6: Upper left corner of randomly generated matrix

8 Results

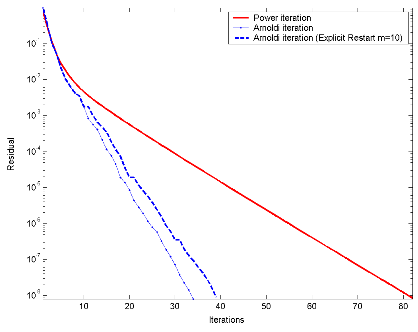
For numerical experiments we have implemented our algorithms in C. To see and compare that our programs runs correctly we have also implemented them in MATLAB. PageRank implemented in MATLAB using the Power method and the normal Arnoldi method can be seen in the Appendix A.1. The Power method and the normal Arnoldi method has also been parallelized using the message-passing interface (MPI).

8.1 The number of needed iterations

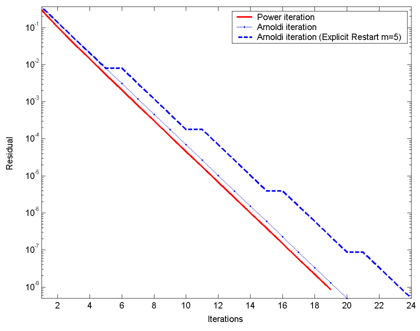
The number of iterations required by the different methods varies greatly. Our results show that the Power method needs more iterations (compared to the Arnoldi method) for convergence. However the structure of the matrix influenced the results. The most important factor to compare between the methods, is the time it takes to calculate pagerank, as an iteration is quite different in the various algorithms. Figure 7 presents results for our three test matrices.



(a) Stanford Web Matrix



(b) it.uu.se matrix



(c) Randomly generated matrix

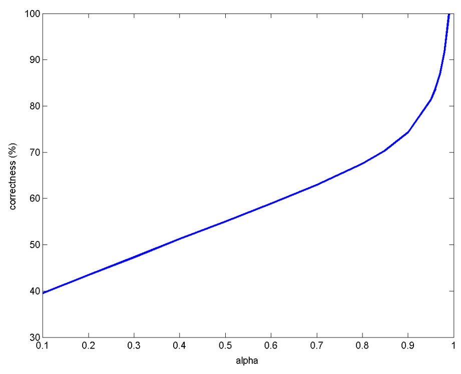
Figure 7: Residual Vs iterations for our three matrices ($\alpha = 0.85$)

We notice that the explicitly restarted Arnoldi method needs a few more iterations than the normal version. One can clearly see where the restarts of the explicitly restarted Arnoldi method occurs. Both the Power and Arnoldi method converge linearly for the randomly generated matrix in Figure 7(c).

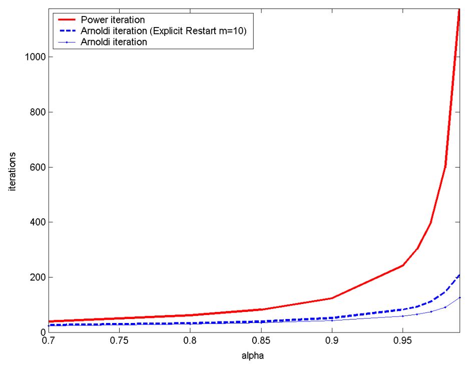
8.2 Varying α

Here we test influence varying α (the teleportation probability).

For each of the matrices we have computed a "correct" PageRank using an α of 0.99 and a residual tolerance of $1e^{-8}$. The calculated PageRank for each different α has been compared with the "true" one. A plot showing the number of iterations needed for our methods as α grows has also been created. We only show the results for the it.uu.se matrix (see Figure 8).



(a) Correctness of PageRank as α grows



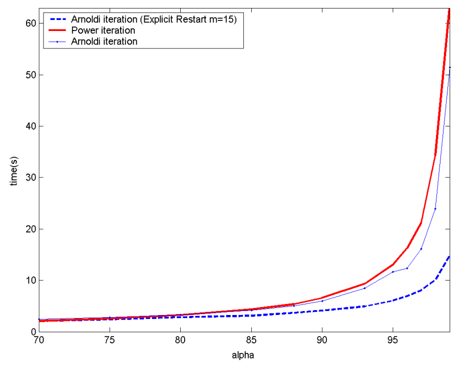
(b) Number of iterations as α grows

Figure 8: it.uu.se matrix

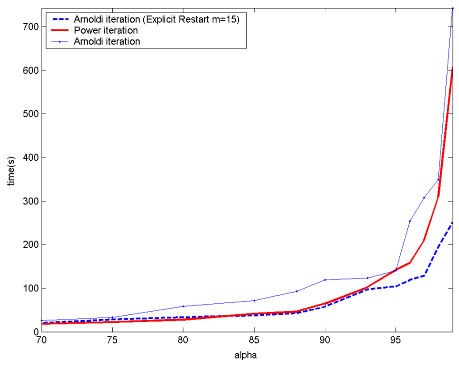
In Figure 8(a) we note that the "correctness" increases linearly up to about $\alpha = 0.85$, after which it seems to increase exponentially.

Figure 8(b) tells us that our two Arnoldi methods seems to handle larger α-values much better than the Power method. We can see this as the number of iterations for the Power method increases much faster than for the Arnoldi methods. Figure 8(b) also tells us that the difference in the number of iterations between the Arnoldi method and the restarted Arnoldi method is insignificant. The small difference between the two Arnoldi methods as well as the extreme increase of number of iterations for the Power method leads us to believe that the restarted Arnoldi method should outperform the other two methods for large values of α.

Figure 9 shows our results of comparing the methods and using different α-values.



(a) it.uu.se matrix



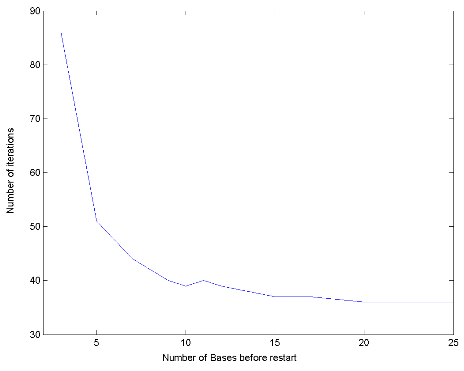
(b) Stanford Web Matrix

Figure 9: Time of PageRank computations as α grows

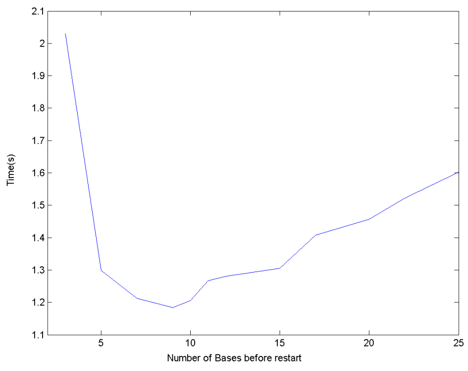
The plots show that the restarted Arnoldi method outperforms the other two methods.

8.3 The importance of m in Explicitly Restarted Arnoldi

By varying the parameter m (max number of bases) in the explicitly restarted Arnoldi method we can change the number of needed iterations and thus also the excecution time. Figure 10 visualizes the importance of chosing a good m value.



(a) Number of iterations



(b) Time

Figure 10: it.uu.se matrix with different m ($\alpha = 0.85$)

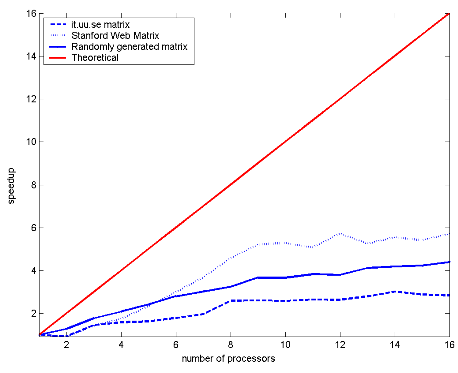
Figure 10(a) shows us that increasing m decreases the number of iterations needed to converge. Thus with a larger m the workload increases as described in section 3.2.3.

Figure 10(b) shows us that for the it.uu.se matrix and an α set to 0.85, m=9 would be the best choice. The size of the best m depends on the matrix structure and thus on the value we have chosen for α. Our tests have shown that an m in the span 5-20 is a good rule of thumb.

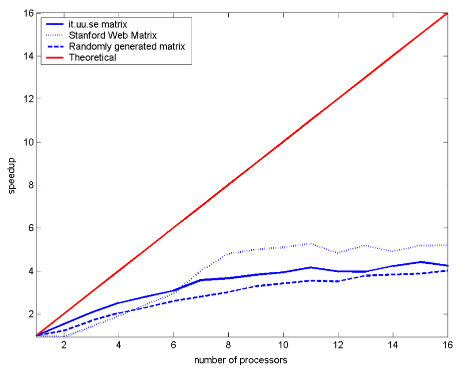
8.4 Parallelization

8.4.1 Speedup

Speedup is a measure of the performance gain achieved by parallelizing an application over a sequential implementation. The definition of speedup is $S_p = T_1/T_p$, where T_1 is the time taken on one processor and T_p is the time taken on p processors. Since we saw that our implementations showed *superlinear speedup*[10] we plot the *absolute speedup* instead. In absolute speedup we use the best single-processor implementation as T_1 above, instead of using the parallel implementation on a single processor.



(a) Power not using load balanced data



(b) Power using load balanced data

Figure 11: Absolute speedup for our three test matrices using the Power method

Figure 11 visualizes the absolute speedup of our parallel implementation of the Power method. Comparing Figure 11(a) and Figure 11(b) one notice that the it.uu.se matrix scales better using load balancing than without. The it.uu.se matrix was the only one of our test matrices where the load balancing had any effect on, see section 8.4.2 for more information on the effect of load balancing.

[10]Superlinear speedup is when the speedup is greater than p.

Figure 12 shows how our parallel Arnoldi implementation scales with the number of processors.

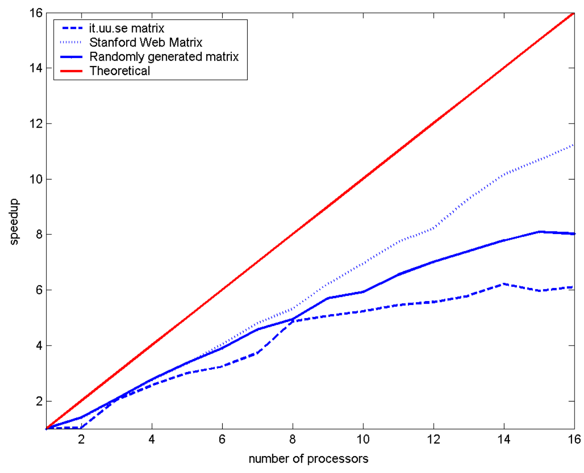
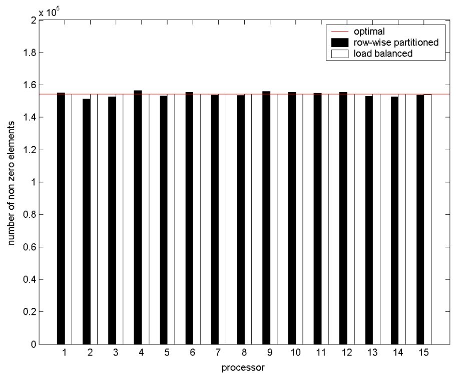


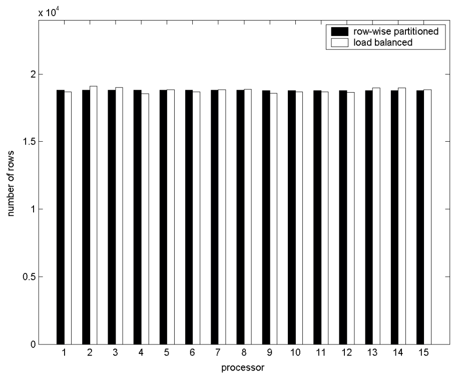
Figure 12: Absolute speedup using the Arnoldi method

It seems that our implementation of the Arnoldi method scales better than the Power method. The reason is probably that our single processor version of the Power method algorithm is better than our single processor version of Arnoldi.

8.4.2 The effect of Load Balancing



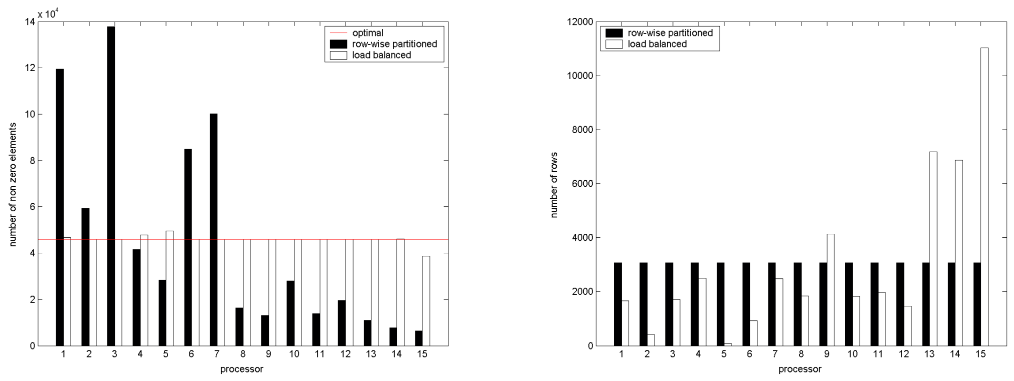
(a) number of non zero elements



(b) number of rows

Figure 13: Load balancing results - Stanford Web Matrix

In Figure 13 we notice that using the load-balancing algorithm, for making sure that each processor has the same number of rows, doesn't change much. As the matrix itself is very well spread out, there are no large unbalanced parts which would give significant improvements if we balance.



(a) number of non zero elements

(b) number of rows

Figure 14: Load balancing results - it.uu.se matrix

Load balancing the crawled matrix is shown in Figure 14. The results show that there are large differences between load balancing or not load balancing. If we view the structure of the matrix (see section 7.2) we see that there are many rows in consecutive order with very few entries per row. This gives large imbalances in the number of non zero elements if we do not load-balance. This matrix is the one where load balancing has the largest effect.

For the randomly generated matrix the difference between balancing or not is negligible, as one would expect since it lacks any real structure.

9 Discussion

How do the investigated methods compare to each other for computing PageRank? The Power method, used by Google, generally works very well for $\alpha < 0.9$. Although it takes a lot of iterations to finish, a good characteristics of this method is that every iteration is as fast as the previous one. The Arnoldi method on the other hand, has an increasing workload and memory requirement by each iteration, which means that too many iterations will be devastating. The explicitely restarted Arnoldi method is a far better method with smaller memory requirements and better speeds.

Our results demonstrate that the Power method takes about the same time as the Arnoldi methods for small α on most matrices. The Power method is therefore preferable under these conditions as it demands far less memory. Results also show that the explicitely restarted Arnoldi method is preferable to both the Power and the Arnoldi methods at higher values of α.

The reason for the Power method's problems at higher selections of α is that the number of neccesary iterations before convergence grows at an exponential rate, a behaviour that doesn't apply to the Arnoldi methods.

To effectively use the restarted Arnoldi method the best selection of m (i.e. the number of bases before restart) needs to be investigated seperately for each matrix one want to use. Since this investigation is not possible for any live system, a good rule of thumb is to use an m-value between 5 and 20.

The tests with the parallelization show that the load balancing algorithm used does not seem very neccesary. For any larger link-matrix, the method of having each processor read the same number of rows will produce results that is just as good as if one explicitely demanded each processor to have about the same number of non zero elements. The reason for this is the low number of elements (links) per row (page), and the general randomness of the Internet's hyperlink structures.

10 Acknowledgements

We would like to thank our advisors for their help and support. Our advisors were prof. Lars Eldén from University of Linköping, Dept. of Mathematics and Maya G. Neytcheva from University of Uppsala, Dept. of Scientific Computing.

References

[1] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide.* SIAM, Philadelphia, 2000.

[2] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 33:107–117, 1998.

[3] Sergey Brin, Lawrence Page, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Computer Science Dept., Stanford Univ., Stanford, USA, 1998.

[4] Lars Eldén. Google mathematics. Talk, Dec. 2003.

[5] Kevin Hemenway and Tara Calishain. *Spidering Hacks^{TM}100 Industrial-Strength Tips & Tools.* O'Reilly & Associates, Inc., first edition, Oct. 2003.

[6] Amy N. Langville and Carl D.Meyer. A survey of eigenvector methods of web information retrieval. Technical report, Dept. of Mathematics, North Carolina State Univ., Raleigh, USA, 2003.

[7] Sergio Pissanetsky. *Sparse Matrix Technology.* Academic Press, Inc. (LONDON) Ltd, 1984.

A Appendix

A.1 PageRank: Power method (Matlab)

```
%-----------------------
function [x,res,I] = powerPageRank(A,c,tol)
 n = max(size(A));
 uniform = ones(n,1)/n;
 v = uniform;
 s = uniform;
 I=0;res=1;
 while res > tol
  [y,res] = iteratePageRank(P,alpha,v,x,res)
  I = I + 1;
 end

%-----------------------
function [y,res] = iteratePageRank(P,alpha,v,x,res)
 % one pagerank iteration
 y = P*x;
 y = alpha*y;
 d = 1 - norm(y,1);
 y = y + d*v;
 res = residual(y,x);
```

A.2 PageRank: Arnoldi method (Matlab)

```
%-----------------------
function [x,res,I] = arnoldiPageRank(A,maxbases,alpha,tol)
disp('Started Arnoldi PageRank');

% Check which columns of the matrix that are all 0s
d = check_empty_columns(A);

% Create the bases
[V,H,res,I] = create_arnoldi_base(A,maxbases,alpha,tol,d);

[EVEC,EVAL ] = eig(full(H));

%Get the biggest eigenvalue and its index
[eigval ind] = max(diag(EVAL));

%Get first eigenvector of H
firstvec = EVEC(:,ind);

%Retrieve first eigenvector
eigvec = V*firstvec;

%Normalize it
eigvec = eigvec./norm(eigvec,1);

%If negative component, "abs" it
if(eigvec(1)<0) eigvec = abs(eigvec); end

x = eigvec;

%-----------------------
function [d] = check_empty_columns(A)
  d = sparse(size(A,1),1);
  for i = 1:size(A,2)
      if sum(A(:,i))==0
          d(i) = 1;
      end
  end

%-----------------------
function [V,h,res,I] = create_arnoldi_base(A,maxbases,alpha,tol,d)
n = size(A,1);
h = sparse(zeros(maxbases+1,maxbases+1));

% Initial guess of vector
V = ones(n,1)/n;

% Normalize starting vector
V = V/norm(V,2);

% Start iteration
for j = 1:maxbases

    w = SpMxV(A,V(:,j),alpha,d);

    for i = 1:j
        h(i,j) = w'*V(:,i);
        w = w - h(i,j)*V(:,i);
    end

    h(j+1,j) = norm(w,2);

    if h(j+1,j) < 1e-12;
        fprintf(1,'w has \"vanished\": %g',h(j+1,j))
        break
    end

    vtemp = w/h(j+1,j);
    V = [V vtemp];

    % computing cheap residual
    [EVEC,EVAL] = eig(full(h(1:j,1:j)));
    [tmp ind] = max(diag(EVAL));
    cheapres = h(j+1,j)*abs(EVEC(j,ind));

    fprintf(1,'iter:%d res=%g \t(tol=%g)\n',j,cheapres,tol)

    if cheapres < tol;
        break
    end
end

res = cheapres;
I = j;

if j==maxbases
    h = h(1:j+1,1:j+1);
else
    h = h(1:j,1:j);
end

%-----------------------
function [y] = SpMxV(Q,z,alpha,d)
  n = size(z,1);
  e = ones(n,1);

  y = alpha*Q*z;
  beta = alpha*d'*z + (1-alpha)*e'*z;
  y = y + beta*e/n;
```

A.3 Web-Crawler (PERL)

```
#!/it/sw/gnu/bin/perl -w
#
# Authors:  PerAnders Ekstrom
#           Erik Andersson
#
# Adapted from hack #46 in "Spidering Hacks"
#
# program parses through a domain looking for valid links.
# if a valid link cannot be downloaded it will still be indexed
# but of course it will not have any outlinks.
#______________________________________________________________#


#--------START OF MAIN-------#

use strict;
use Getopt::Std;
use LWP::UserAgent;
use HTML::LinkExtor;
use URI::URL;

my $help = <<"EOH";
------------------------
TDB crawler.

Options: -l start link
         -d domain
         -f filename
         -m matlab format
         -h help

Example:

  ./tdbcrawler.pl -l http://www.it.uu.se -d it.uu.se -f ./it.uu.se -m

------------------------
EOH

# declare global variables
my %args;           # hash of input arguments
my $domain;         # domain user want to crawl
my $filename;       # output filename
my %todo;           # hashes of arrays with URLs to parse
my %done;           # hashes of arrays with parsed/finished URLs
my $bytes = 0;      # bytes downloaded
my $id;             # id of each link
my $url;            # global variable URL
my $filter = "\/|htm|html|dhtml|xhtml|shtml|asp|aspx|php|php3|phps";
my $crawler_name = 'TDBCrawler';

# parse the input arguments
getopts("l:d:f:m:h",\%args);
die $help if exists $args{h};

if(exists $args{'l'} && exists $args{'d'} && exists $args{'f'})
  {
    $domain = $args{d};
    $filename = $args{f};
  }
else {  die $help;  }

if(exists $args{'m'}) {  $id=1;  }  # (matlab)
else {                   $id=0;  }  # (crs)


print "=" x 80 . "\nGoing to spider domain '$domain'";
print " starting at link $args{l}\n" . "=" x 80 . "\n";

# push our first link into the todo list
$todo{$args{l}}[0] = $id++;

# start spidering
walksite();

# create sparse matrix out from the hashes and save to file
saveSpM();

# save links to file
saveLinks();

$bytes = $bytes/1000;
print "Downloaded: $bytes kb of data\n";
print "-" x 80 . "\n";

#------------END OF MAIN------------#
```

```
#--------------
# sub walksite:
#      Dequeues and fetches all urls listed in %todo
#      Parses the pages and saves their links in %todo
#      When done with a url enqueue it to %done
#-------------------------------------------------
sub walksite
  {
    do
      {
        # get first URL to do.
        $url = (keys %todo)[0];

        # download this URL.
        my $browser = LWP::UserAgent->new;
        $browser->timeout(1);
        my $resp = $browser->get($url,'User-Agent'=>$crawler_name);

        # check the results.
        if($resp->is_success)
          {
            my $base = $resp->base || '';
            my $data = $resp->content;

            # increase our bytes counter
            $bytes = $bytes + length($data);

            HTML::LinkExtor->new(\&findurls,$base)->parse($data);
          }
        else
          {
            # couldn't download URL
            print "$url couln't be downloaded\n";
          }

        # we're finished with this URL, so move it from the TODO list
        # to the DONE list, (and print a report).
        $done{$url} = $todo{$url};
        delete $todo{$url};

        print "-> processed URLs: " . (scalar keys %done) . "\n";
        print "-> remaining URLs: " . (scalar keys %todo) . "\n";
        print "-" x 80 . "\n";
      } until ((scalar keys %todo) == 0);
  }

#--------------
# sub findurls:
#      in->link
#      if link already exists push it to list of referred url
#      elsif link pass our filters: add it to %todo
#---------------------------------
sub findurls
  {
    my($tag, %links) = @_;
    return if $tag ne 'a';
    return unless $links{href};

    # already seen this URL, its in our done list.
    if(exists $done{$links{href}})
      {
        push(@{$todo{$url}}, $done{$links{href}}[0]);
        return;
      }
    # already seen this URL, its in our todo list.
    if(exists($todo{$links{href}}))
      {
        push(@{$todo{$url}} ,$todo{$links{href}}[0]);
        return;
      }

    # OK, havn't seen this URL, run through our filter.
    if( $links{href} =~ /(\S)*($domain)(\S)*($filter)+$/ )
      {
        # add index of link which we point at
        push(@{$todo{$url}},$id);

        # increase our outlinks counter
        $todo{$links{href}}[0] = $id++;
       }
  }

#--------------
# sub saveSpM:
#      saves link-structure in either Matlab- or CRS-format
#----------------------------------
```

```
sub saveSpM
  {
    my $tmp;
    my $matrixfile = "$filename.matrix";

    open(FP, ">$matrixfile");

    if(exists $args{'m'})  # Matlab
      {
        print "Writing (matlab) to file: $filename\n";
        for my $urls (keys %done)
          {
            if($#{$done{$urls}}>0)  # if have outlinks
              {
                my $ind = $done{$urls}[0];
                my $val = 1/$#{$done{$urls}};

                for my $i ( 1 .. $#{$done{$urls}})
                  {
                    print FP "$ind $done{$urls}[$i] $val\n";
                  }
              }
          }
      }
    else      # Compressed Row Storage (CRS)
      {
        print "Writing (CRS) to file: $filename\n";
        # dimension
        my $dim = scalar(keys(%done));

        # write nnzero
        my $nnzero = 0;
        for my $urls (keys %done)
          {
            for my $i ( 1 .. $#{$done{$urls}} )
              {
                $nnzero++;
              }
          }
        print FP "$dim $dim $nnzero\n";

        # write row_ptr
        my $row_ptr = 1;
        print FP "$row_ptr ";
        for my $urls (keys %done)
          {
            $row_ptr += $#{$done{$urls}};
            print FP "$row_ptr ";
          }
        print FP "\n";


        # write col_ind
        for my $urls (keys %done)
          {
            for my $i (1 .. $#{$done{$urls}} )
              {
                print FP "$done{$urls}[$i] ";
              }
          }
        print FP "\n";

        # write val
        for my $urls (keys %done)
          {
            if($#{$done{$urls}}>0)
              {
                $tmp = 1/$#{$done{$urls}};
                print FP "$tmp " x $#{$done{$urls}};
              }
          }
        print FP "\n";
      }

    close(FP);
  }

#--------------
# sub saveLinks:
#      saves links to file
#      format: <index>: <url> -> <index> <index> ...
#---------------------------------
sub saveLinks
  {
    my $linkfile = "$filename.links";
    open(FP, ">$linkfile");

    # print links to file
    foreach my $urls (keys %done)
      {
        print FP "$done{$urls}[0]: $urls => ";
        #foreach my $element (@{$done{$urls}})
        for my $i ( 1 .. $#{$done{$urls}} )
          {
            print FP "$done{$urls}[$i] ";#"$element ";
          }
        print FP "\n";
      }
    close(FP);
  }
```