PAGE RANK IMPLEMENTATION USING OPEN MP AND JAVA MULTITHREADING

Roohi Bharti #009390927
Computer Science Department
San Jose State University
San Jose, CA, USA
roohi.bharti900@gmail.com

Parul Sharma #009432215
Computer Science Department
San Jose State University
San Jose, CA, USA
parulsharma.1592@gmail.com

ABSTRACT

This paper's aim is to propose the implementation of the Page Rank Algorithm using Open Multi Processing Application Programming Interface and Java Multithreading. Page Rank algorithm is used by the Search Engines to give an ordered list of the web search results in response to the user query. Servers producing these results produce the required results faster than the serial implementation of the same.

1. INTRODUCTION

In this paper we plan to implement the Page Rank algorithm in a parallel manner. Page Rank is used by the Google Search Engine to rank the web pages on the web nets. For the enormous web pages and large user requests the computations should be done expediently and effective fashion. This calls for the parallel enact ion of the algorithm. We will be using the Open MP API and Java Multithreading to do the same. Open MP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the super computer. We further plan to conclude and summaries the results of the analysis.

2. PAGE RANK ALGORITHM

Search Engine responds to the user's request and applies the web mining techniques to filter and classify the results. The classified order of search results is obtained by applying some special algorithms called Page Ranking algorithms. Page Rank is an algorithm invented by Google to measure the relative importance of web pages, which is also called popularity. Ranking is based on the topology of the web. The links structure between the pages decides the rank. The main vision is that if a page A has a link to a page B, then the page A considers that page B is important enough to deserve being voted and might be visited by visitors of page A. This link from A to B increases the Page Rank of B or votes in favor of Page B.

2.1 Overview

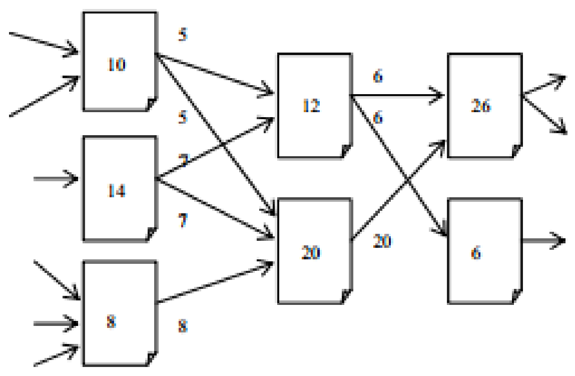
World Wide Web is the repository of information of interlinked hypertext document which is accessible through the internet. Web may contain data in multiple forms like :text, video, images and other multimedia data. The user navigates through this data using hyperlink structure. The sorted order of search results is obtained by applying some special ranking algorithms called Page Ranking algorithm. We will be reviewing its implementation here.

2.2 History

Page Rank is one of the few algorithms that is speculated deeply in the mathematical field that has not only been studied extensively by mathematicians and computer scientists, but is also regularly commented on by scholars of the vast domain. It was invented by Larry Page and Sergey Brin. Both were graduate students at Stanford, and it became a Google standard in 1998. As defined by Google : "Page Rank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The ever most paper about this algorithm explained the Page Rank and the initial prototype of the Google search engine, was published in 1998 which was soon after, Page and Brin founded Google Inc..The underlying assumption is that more important websites are likely to receive more links from other websites" . The core idea behind Page Rank was that the importance of any web page can be determined by looking at the pages that link to it.

2.3 The Web anatomy and Page Ranks

The structure of hyperlinks and links decides the Page rank of a web page. As said earlier if a web page has a vital incoming link to it then this incoming link makes even the outgoing links from that page crucial for page rank computation



.

Figure 1: Distribution of Page Ranks on web

2.4 Page Rank Explained

Quoting from the original Google paper, Page Rank is defined like this:

"We assume page A has pages T1...Tn which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. There are more details about d in the next section. Also C(A) is defined as the number of links going out of page A. The Page Rank of a page A is given as follows:

PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))

The Page Ranks form a spread of probability distribution over web pages, so the sum of all web pages' Page Ranks will be one.

Page Rank or PR(A) can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the web."

The terminology's explanation is as follows:

PR(Tn) - Every page has a conviction of its own self-significance. This means that "PR(T1)" is the vote for the first page in the web all the way up to the vote for the last page on the web ie. "PR(Tn)" .

C(Tn) - Every page strew its vote out evenly amongst all of it's outgoing links. The count which is the number of outgoing links for page 1 is "C(T1)", similarly for Page n the number of outgoing links will be "C(Tn)"

PR(Tn)/C(Tn) – Back links are also counted in the computation of the page rank . If our page (page A) has a back link from page "n" the contributed part of the vote page A will get is "PR(Tn)/C(Tn)"

d(... – It is the damping factor. All the page ranks when added will give the page rank of the desired page but when all these fractions of votes are added together, one page can influence the computation too much. To stop the other pages having too much influence, this total vote is "damped down" by multiplying it by 0.85 (the factor "d").

(1 - d) –The sum of all the web pages' Page Ranks will be one. The (1 – d) bit at the start is a bit of probability math. It adds in the bit lost by the d(....A page having no back link to it will have the OR of 0.15. (i.e. 1 – 0.85).

2.5 How are calculations done

As explained above, the PR of each page depends on the PR of the pages directing to it. Thus to calculate the page rank of a web page we need to compute the page rank of all the web pages that are pointing to that page. As we will not know what PR those pages have until the pages pointing to them have their PR calculated and so on. The cyclic dependency on the calculation of PR seems impossible to compute this.

Google paper states that:

"PageRank or PR(A) can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the web."

This computation is the iterative process. This means that we can just start with the computation of the page rank of a web page without knowing the final value of the page rank of the other pages. Basically, each time we compute the calculation we're getting a closer estimate of the final value of the page ranks. Thus all we need to do is to memorized the page rank values we calculate and repeat the calculations a specific number of times until the numbers reach a stable value.

To analyze what is just said above, lets take the simplest example network: two pages, each pointing to the other:

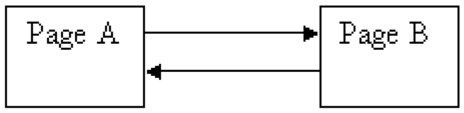


Figure 2: Simple Network of web pages

Each page has one outgoing link. The outgoing count or number is 1, i.e. C(A) = 1 and C(B) = 1. Guess 1

We will begin the calculations making some guesses about the initial rank of a web page. For the first case we can assume the PR at 1.0. The calculations are as follows:

d= 0.85

PR (A) = (1 – d) + d(PR(B)/1)

PR (B) = (1 – d) + d(PR(A)/1)

i.e.PR (A) = 0.15 + 0.85 * 1

= 1

PR (B) = 0.15 + 0.85 * 1

= 1

In the above case the PR values have come to a stability.

Further we can compute the PR of the pages by making another assumption about the initial value of the PR of a web page.

d= 0.85

PR(A)= 0.15 + 0.85 * 0

= 0.15

PR(B)= 0.15 + 0.85 * 0.15

= 0.2775

Now again:

PR(A)= 0.15 + 0.85 * 0.2775

= 0.385875

PR(B)= 0.15 + 0.85 * 0.385875

= 0.47799375

Again

PR(A)= 0.15 + 0.85 * 0.47799

= 0.5562946

PR(B)= 0.15 + 0.85 * 0.5562946

= 0.6228504

This can be repeated a predetermined number of times.

For the next set of calculations let's make the initial assumption at 40 each. The calculations will be as follows:

PR(A) = 40

PR(B) = 40

First iteration will be:

PR(A)= 0.15 + 0.85 * 40

= 34.25

PR(B)= 0.15 + 0.85 * 0.3858

= 29.17

For next iteration,

PR(A)= 0.15 + 0.85 * 29.177

= 24.95087

PR(B)= 0.15 + 0.85 * 24.95087

= 21.35824

The calculations show that the numbers are increasing. Numbers will reach 1.0 and then will stop increasing.

We can make a conclusion that it doesn't matter where you start your assumption as after certain number of iterations the Page Rank calculations comes to stability which means the "normalized probability distribution" will be 1.0.

2.5.1 Number of Iterations

Decision about the number of iterations is the trivial question. For the network which is enormous as the World Wide Web there can be many millions of iterations possible. Damping factor helps to settle down the calculations to the stability. If the damping factor is too high then it takes large number of iterations for the numbers to adjudicate the ground, if it's too low then you then the results will be oscillating both above and below the average and might not settle down.

The ordering of the calculations does not matter. The answer will always come out the same no matter which order you choose. The only fact is that some orders will get you there quicker than others.

2.5.2 Network Examples

Consider the example of simple websites based on below formula and below example of network of web pages . The nodes represent the web pages and the edges represent the links between them.The computation of the PR for the web pages is as follows :

PR(A) = (1-d) + d (PR(T1)/C(T1) + ... + PR(Tn)/C(Tn))

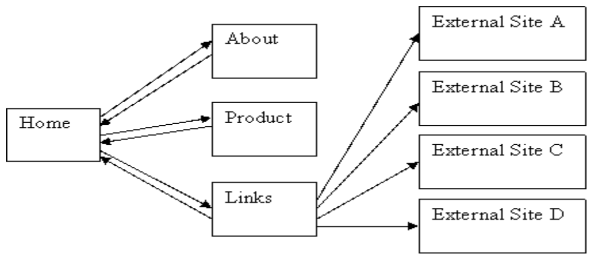


Figure 3: A simple hierarchy of network.

Upon following the calculations the final PR of the web pages are:

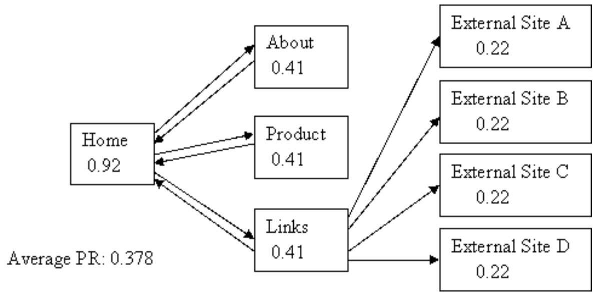


Figure 4: PR for the simple hierarchy of network

Average PR for such a system came out to be: 0.378.

For a system like below the PR of the web pages will be computed to following values:

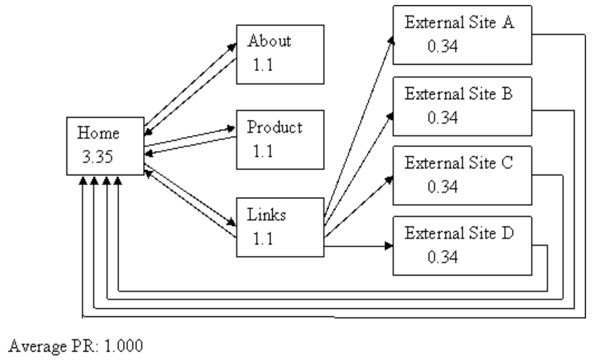


Figure 5:A meshed network.

For the system containing loops the PR will be as follows:

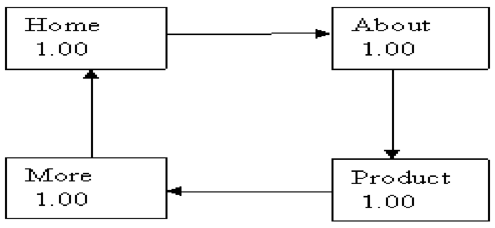


Figure 6: A looped network.

2.6 Serial Implementation of Page Rank

We start by modeling the Web net as a directed graph, with nodes represented by web pages and edges represented by the links between them. Consider the following system containing four web pages:

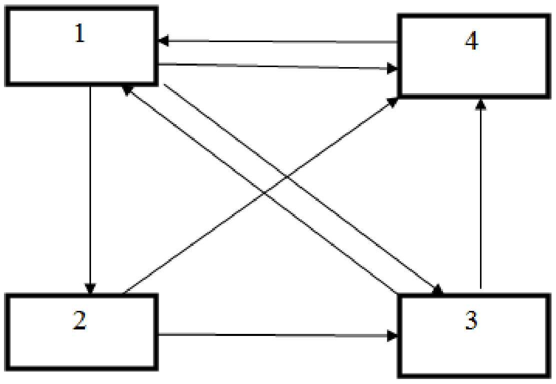


Figure 7: Implementation Model.

.The connectivity in the system is defined as follows:

1. Web page 1 has the links for the Web pages 2, 3 and four.
2. Web page 2 has the link to web pages 3 and 4.
3. Web page 3 has the link to the web page 1.
4. Web page 4 has the links to web page 3 and 1.

2.6.1 Weighted edges of the Web Nets

In our analysis ,our model follows the notion that each page should transfer evenly its importance to the pages that it links to. Node 1 has 3 outgoing edges, so it will pass on of its importance to each of the other 3 nodes. Node 3 has only one outgoing edge, so it will pass on all of its importance to node 1. In general, if a node has m outgoing edges, it will pass on of its importance to each of the nodes that it links to. The visualization of the same is depicted below:

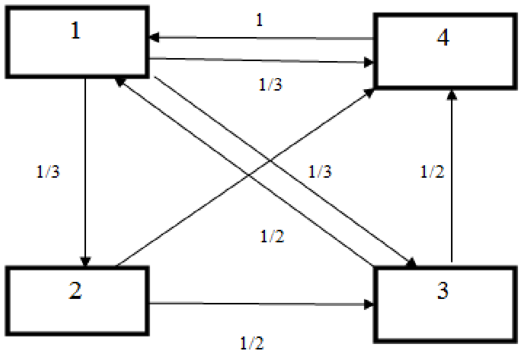


Figure 8: Network Topology and links.

2.6.2 Transitional Matrix Notations

The web net can be translated into a matrix which contains the weight age of the links on the overall system of web pages represented in the form of a matrix A:

$$\begin{bmatrix} 0 & 0 & 1 & 1/2 \\ 1/3 & 0 & 0 & 0 \\ 1/3 & 1/2 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

2.6.3 Dynamic Behavior of the Web Nets

Let us assume that initially the importance of the web pages is evenly distributed among the 4 nodes thus each getting ¼. The initial rank matrix which is denoted by v have all the entries equal to ¼. The importance of the web page is increased with every incoming link coming to a web page Thus to start at the first step, we will update the rank of each page by adding to the current value the importance of the incoming links. This addition operation is similar to multiplying the matrix A with v. At step 1, the new rank vector is v1 = Av. We will follow the number of iterations. At step 2, the updated rank vector is v2 = A(Av) =A^2v. Numeric computations are as follows:

$$v = \begin{pmatrix} 0.25 \\ 0.25 \\ 0.25 \\ 0.25 \end{pmatrix},\ Av = \begin{pmatrix} 0.37 \\ 0.08 \\ 0.33 \\ 0.20 \end{pmatrix},\ A^2 v = A\,(Av) = A \begin{pmatrix} 0.37 \\ 0.08 \\ 0.33 \\ 0.20 \end{pmatrix} = \begin{pmatrix} 0.43 \\ 0.12 \\ 0.27 \\ 0.16 \end{pmatrix}$$

$$A^3 v = \begin{pmatrix} 0.35 \\ 0.14 \\ 0.29 \\ 0.20 \end{pmatrix},\ A^4 v = \begin{pmatrix} 0.39 \\ 0.11 \\ 0.29 \\ 0.19 \end{pmatrix},\ A^5 v = \begin{pmatrix} 0.39 \\ 0.13 \\ 0.28 \\ 0.19 \end{pmatrix}$$

$$A^6 v = \begin{pmatrix} 0.38 \\ 0.13 \\ 0.29 \\ 0.19 \end{pmatrix},\ A^7 v = \begin{pmatrix} 0.38 \\ 0.12 \\ 0.29 \\ 0.19 \end{pmatrix},\ A^8 v = \begin{pmatrix} 0.38 \\ 0.12 \\ 0.29 \\ 0.19 \end{pmatrix}$$

Figure 9: Page Rank Computation

The ranks of the web pages come to stability after eight iterations. This can be considered as the final rank of the four web pages. The final matrix can be considered the page rank vector the web net.

2.6.4 Probabilistic View of the Web Nets

The popularity is the core quantum which is used to measure the importance of a web page. We have perceived the notion of the importance of a page 'I' as the probability that a casual surfer on the Internet who uses the web browser and starts following hyperlinks, visits the page I. We can contemplate the weights we given to the to the edges of the graph (refer to Figure 8) in a probabilistic way: A casual surfer that is currently on the web page 2, will have the ½ probability to go to page 3, and ½ probability to go to page 4. This can be perceived as the random walk on the web nets. Each page has equal probability ¼ to be chosen as a starting point. The column vector [¼ ¼ ¼ ¼]t will be the initial rank vector. The probability that page I will be visited after one step is equal to Ax, and so on. These calculations will be converged to the probabilistic vector v*. This vector v* is called the stationary distribution and it will be our Page Rank vector.

2.6.5 Implementation of Page Rank in C Language

After analyzing the numerical computations, we have implemented the algorithm in C language in the serial manner. We have assumed a web net of four pages and computed the ranks in the similar fashion as mentioned above. The Initial Rank vector will contain the initial ranks for the web pages in the web nets. For initial computation we have considered the equal ranks for all the four web pages. The Transitional Vector contains the information about the links in between the web pages. These two matrix are multiplied for some fixed number of iterations after which the system reaches the equilibrium. The final output of the algorithm is the column matrix which contains the final ranks of the web pages.

3. NEED FOR PARALLEL EXECUTION

The need for parallel execution arises from the problems which are "too" expensive to be solved with the classical approach. The need of computation of the results on specific or reasonable time The need of results on specific (or reasonable) time.

The serial implementation of such an algorithm will be slow. Since this algorithm is implemented for the purpose of speedy computation of the results such as search engine computation of the ranks for the web pages, the parallel implementation of the computations will be desirable. The pursuit of Page Rank algorithm has sequences of code which can be executed in parallel.

4. PARELLELISM THROUGH OPEN MULTI-PROCESSING

Structured parallelism can be enacted with the help of Open MP in various applications. With the hike in the convoluted complexity of the working of the applications, there is the requirement for the irregular implementation of the parallelism. Intrinsic goals of OpenMP were to define a benchmark patios to express and to maneuver unstructured parallelism in the code effectively. By providing the simple and flexible interface for developing portable and scalable parallel applications, OpenMP is classified as an efficient standard for shared-memory parallel programming. OpenMP in the 1990s due to the requirement of a standard vendor specific directives related to parallelism. OpenMP was organized around parallel loops constructs and was structured to handle dense numerical applications.

4.1 History

The OpenMP Architecture promulgate the first API for Fortran in October 1997. In forthcoming years they released the versions for the C/C++. In year 2000 Open MP released the second version for Fortran and C/C++. The following table presents the versions of the Open MP version and support for different languages.

Table 1. Open Mp version Compatibility

Year	OpenMP Version	Fortran	C/C++
1997	1.0	Yes	Yes
2000	2.0	Yes	No
2002	2.0	No	Yes
2005	2.5	Yes	No
2008	3.0	Yes	Yes
2011	3.1	Yes	yes
2013	4.0	Yes	Yes

The OpenMP standard specification commenced in the spring 1997, taking over where ANSI X3H5 has left off.

There are certain partners in the OpenMP Standard specification included:

- Compaq
- Hewlett Packard
- Intel corporation
- Sun Microsystems, Inc.
- U.S Department of Energy ASC Program

4.2 Goals of OpenMP

There are many goals of OpenMP to attain which embrace the followings:

Standardization

OpenMP provides the levelness betwixt a variety of shared Memory Architectures/platforms.

Efficient

It establish a simple and limited set of directives for programming shared memory machines. Eloquent parallelism can be implemented by using 3 to 4 directives.

Elementary

It has the capability to incrementally parallelize a serial program, which is not similar to other message passing libraries. It also has the capability to implement both coarse grain and fine grain parallelism.

4.3 Core Elements

The Core Elements of the OpenMP are the constructs for User-level runtime routines, thread creation, workload distribution and data environment.

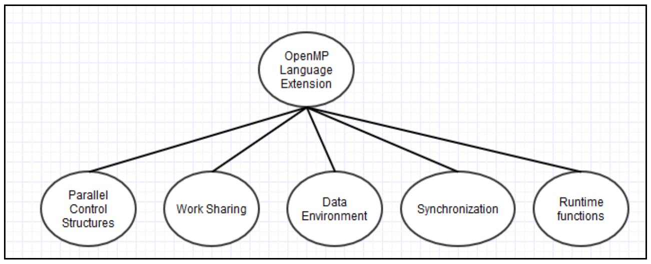


Figure 10:Core Elements of Open MP

4.4 OpenMP Programming Model: Shared memory model:

OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. We can depict the same with the help of the diagram which will be useful for UMA and NUMA. UMA- Uniform Memory access architecture is as follows:

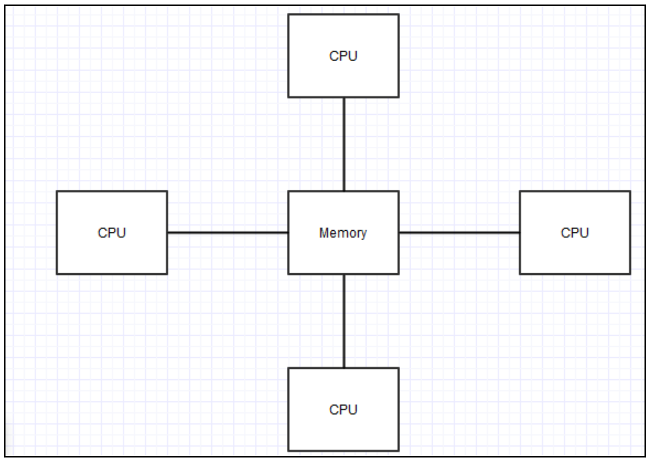


Figure 11: UMA Model for Shared Memory

NUMA- Non Uniform Memory access architecture is as follows:

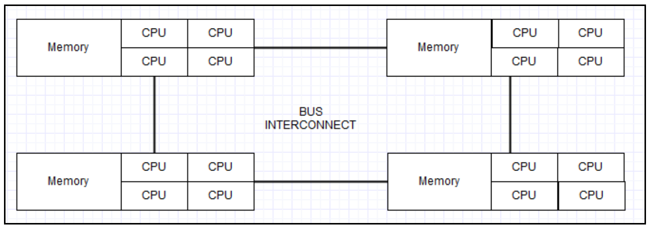


Figure 12: NUMA Model for Shared Memory

4.5 ARCHITECTURE DETAILS

4.5.1 Thread Based Implementation

Open MP attain parallelism through the use of the threads. We know that thread is the basal unit of processing that can be scheduled by an operating system. As is the custom, the number of threads will match the number of the machines processors and cores.

4.5.2 Explicit Parallelism

For the programmer to get the full bridle over parallelization, OpenMP will be used as an explicit programming model. We may simplify parallelization by saying that here we have to feed serial program to the compiler directives, at the same time we can make complex as inserting subroutines to set multiple levels of parallelism, locks and even nested locks.

4.5.3 Fork and Join Model

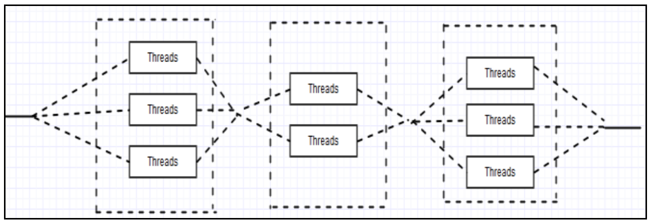


Figure 13: Open MP-Fork and Join Model

OpenMP uses the **fork-join** model of parallel execution:
• All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered.
• **FORK**: the master thread then creates a *team* of parallel threads
• The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads
• **JOIN**: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread.

4.5.4 Compiler Directive Based:

Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

4.5.5 Nested Parallelism Support

The API provides for the placement of parallel constructs inside of other parallel constructs. Implementations may or may not support this feature.

4.5.6 I/0 Features

- OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

4.5.7 Flush Often

OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory. In other words, threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time. When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSH by all threads as needed.

4.5.8 General Rules:

Following are the general rules that we should follow:
1. Case Sensitive
2. Directives follow conventions of the C/C++ standards for compiler directives

3. Each directive applies to at most one succeeding statement, which must be a structured block.

5. OPENMP API OVERVIEW

The API includes three main components:

1. The OpenMP API is compromised of three distinct components:

- Environment Variables
- Runtime Library Routines
- Compiler Directives

2. The application developer decides how to employ these components.

3. Implementation is different for all API Components.

5.1 Compiler Directives

Compiler directives appear as comments in your source code and are ignored by compilers unless you tell them otherwise - usually by specifying the appropriate compiler flag.

- Spawning a parallel region
- Dividing blocks of code among threads
- Distributing loop iterations between threads
- Distributing loop iterations between threads
- Synchronization of work among threads

5.1.1 Runtime Library Routines

The OpenMP API includes an ever-growing number of run-time library routines.

Various uses:

- Setting and querying the number of threads
- Querying a thread's unique identifier (thread ID), a thread's ancestor's identifier, the thread team size
- Setting and querying the dynamic threads feature
- Querying if in a parallel region, and at what level
- Setting and querying nested parallelism
- Setting, initializing and terminating locks and nested locks
- Querying wall clock time and resolution

5.1.2 Environment Variables

OpenMP provides several environment variables for controlling the execution of parallel code at run-time.

Uses:

- Setting the number of threads
- Specifying how loop interations are divided
- Binding threads to processors
- Enabling/disabling nested parallelism; setting the maximum levels of nested parallelism
- Enabling/disabling dynamic threads
- Setting thread stack size
- Setting thread wait policy

5.2 Fortran Directive Format

Let's discuss about the format which states that it is case sensitive

Table 2:Fortran Directives

Sentinel	All Fortran OpenMP directives must begin with a sentinel. The accepted sentinels depend upon the type of Fortran source. Possible sentinels are: **!$OMP** **C$OMP** ***$OMP**
directive-name	A valid OpenMP directive. Must appear after the sentinel and before any clauses.
Clause	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.

We can also cite one example of it

```
!$OMP PARALLEL DEFAULT(SHARED)
PRIVATE(BETA,PI)
```

- Several Fortran OpenMP directives come in pairs and have the form shown below. The "end" directive is optional but advised for readability.
- Comments can not appear on the same line as a directive
- Fortran compilers which are OpenMP enabled generally include a command line option which instructs the compiler to activate and interpret all OpenMP directives.

5.3 C/C++ Directive Format

Format is as follows:

Table 3:C/C++ Directives

#pragma omp	Required for all OpenMP C/C++ directives.
Directive-name	A valid OpenMP directive. Must appear after the pragma and before any clauses.
Clause	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.
Newline	Required. Precedes the structured block which is enclosed by this directive.

There are general rules to it as well:

- Case sensitive

- Directives follow conventions of the C/C++ standards for compiler directives
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

5.4 Work sharing constructs

As the name states, this constructs basically divide the task among different members. This construct will not create any of the new threads. There is no entail sequence on entry to a work-sharing construct, however end there is barrier.

Types of Work-Sharing Construct:

5.4.1 Loops Do.For:

It shares iterations of a loop across the team. It represents a type of "data parallelism".

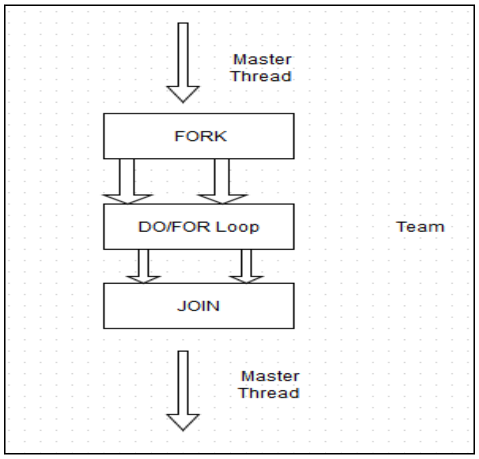


Figure 14: Do/For Loop Contructs

5.4.2 Sections Constructs

It breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism"

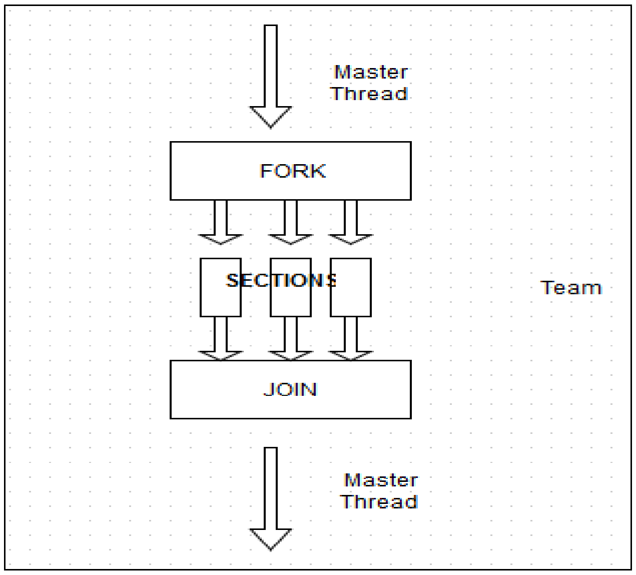


Figure 15: Sections Contructs

5.4.3 Single Thread

Serializes a section of code

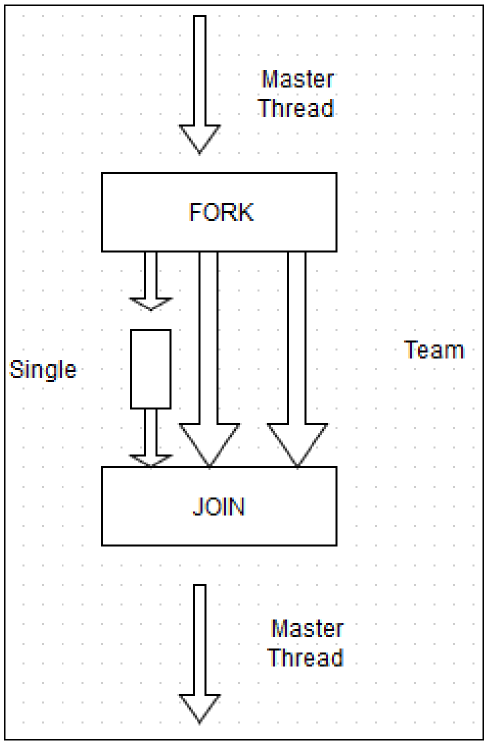


Figure 16:Single Thread Construct

5.4.4 Restrictions

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all

- Successive work-sharing constructs must be encountered in the same order by all members of a team.

6. PARALLEL IMPLEMENTATION OF THE PAGE RANK ALGORITHM

The page rank algorithm involves the matrix multiplication of the transition matrix and the initial rank matrix. The matrix multiplications can be done in parallel provided the computations do not involve any data dependencies. The compiler directives are added to the serial code to parallelize the flow.

The functions used are included in the header file as follows:

#include<omp.h>

The Open MP constructs are applied to the structured code. The structured code is the one which includes the one point of entry and has one point of exit at the bottom of the code.

The environment variables of Open MP will control the runtime execution. Most of the constructs in Open MP are the compiler directives given by the following statement. The additional threads to carry out the task which is surrounded in the parallel constructs, the pragma omp parallel is used. We can specify the number of threads in the pragma clause.

#pragma omp construct [clause [clause]…]

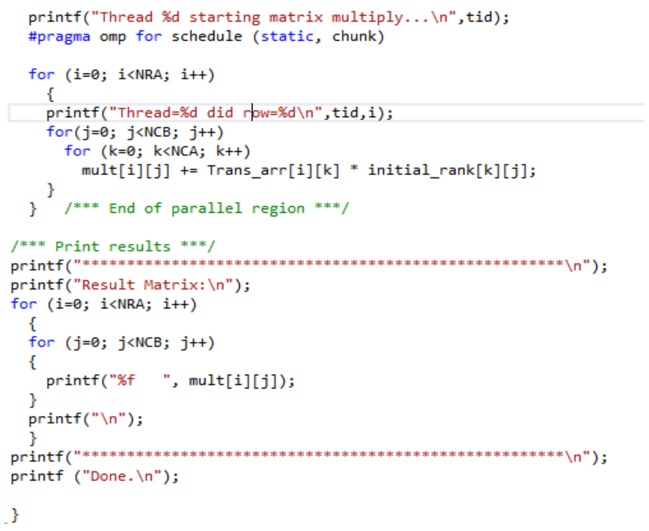
Following library.function returns ID of the thread executing the code.

omp_get_thread_num();

We can set the number of threads which will be executing the code at the run time using the following function:

omp_set_num_threads(n);

The core feature in the implementation of the Page Rank in the parallel manner is the matrix multiplication. Below is the code snippet for the same:



```
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)

  for (i=0; i<NRA; i++)
    {
    printf("Thread=%d did row=%d\n",tid,i);
    for(j=0; j<NCB; j++)
      for (k=0; k<NCA; k++)
        mult[i][j] += Trans_arr[i][k] * initial_rank[k][j];
    }
  }   /*** End of parallel region ***/

/*** Print results ***/
printf("******************************************************\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
  {
  for (j=0; j<NCB; j++)
  {
    printf("%f   ", mult[i][j]);
  }
  printf("\n");
  }
printf("******************************************************\n");
printf ("Done.\n");

}
```

Figure 17: The code snippet for the algorithm using Open MP

The compilers have to be scheduled for handling and balancing the amount of work to be done by a processor. This type of scheduling is called Guided Scheduling. There are two types of scheduling: Static and Dynamic. In static scheduling, the work is divided evenly amongst all the processors available. Whereas in dynamic scheduling the work load is distributed unevenly amongst the processors. The chunk size is the amount of work done per iteration in the parallel version of the serial code. The chunk size can be set by the programmer of the application.

For this implementation the number of threads is decided dynamically at the run time by the API.

The above mentioned code is the prototype of the Page Rank implementation using Open MP. The matrix multiplication of the initial rank vector and the transitional vector is repeated certain number of times till the time the web net comes to equilibrium of the Page Ranks.

7. JAVA MULTITHREADING

Back in the old days a computer had a single CPU, and was only capable of executing a single program at a time. Later came multitasking which meant that computers could execute multiple programs at the same time. It wasn't really "at the same time" though. The single CPU was shared between the programs. The operating system would switch between the programs running, executing each of them for a little while before switching. Along with multitasking came new challenges for software developers. Programs can no longer assume to have all the CPU time available, nor all memory or any other computer resources. Later yet came multithreading which mean that you could have multiple threads of execution inside the same program. A thread of execution can be thought of as a CPU executing the program. When you have multiple threads executing the same program, it is like having multiple CPU's execute within the same program.

Multithreading is even more challenging than multitasking. The threads are executing within the same program and are hence reading and writing the same memory simultaneously. This can result in errors not seen in a single-threaded program. Some of these errors may not be seen on single CPU machines, because two threads never really execute "simultaneously". Modern computers, though, come with multi core CPU's. This means that separate threads can be executed by separate cores simultaneously. A java application runs by default in one process, within java there are several threads that will work together to achieve parallel processing behavior. A Java program runs in its own process and by default in one thread. Java supports threads as part of the Java language via the Thread code. The Java application can create new threads via this class.

7.1.1 JAVA THREADS AND USAGE

The base means for concurrency are is the java.lang.Threads class. A Thread executes an object of type java.lang.Runnable. Runnable is an interface with defines the run() method. This method is called by the Thread object and contains the work which should be done. Therefore the "Runnable" is the task to perform. The Thread is the worker who is doing this task.

Using thread class directly has certain disadvantages which are as follows:

- Creating a new thread causes some performance overhead.
- Too many threads can lead to reduced performance, as the CPU needs to switch between these threads.

- You cannot easily control the number of threads, therefore you may run into out of memory errors due to too many threads.

A thread is considered to be the single sequence of execution which can refers to multiple threads of control within a single program which means. each program can run multiple threads of control within it.

The idea of the multiprocessing is enacted in a multithreaded program by maintaining threads which are obtained from the pool of available ready-to-run threads and run on the available system's processors. The Operating System scheduler can move threads from the processor to either a ready or blocking queue. When a thread performs such a behavior it is said to have "yielded" by the processor. Also the Java virtual machine (JVM) can govern the thread activity movements. This can be done either in a cooperative or preemptive model. This means that the threads are moved from a ready queue onto the processor. Once getting the resources the thread can begin executing its program code.

The course of action that takes place when we use different types threads. Application Thread will be executed as follows:

- The main() method defines the task of the thread object which is created by Java Virtual Machine.
- Java Virtual Machine starts the thread object.
- The thread will execute the statements sequentially.
- The thread terminates and dies after execution of all the statements and method will return the value to be returned.

If we have multiple threads in an application then the execution will be done as follows**:**

- All threads share the same memory space. Thus each thread has its private run-time stack.
- If two threads execute the same method, each will have its own copy of the local variables the methods uses.
- All threads can access the same dynamic memory, i.e., heap.
- Two different threads can use the same object and same static fields at the same time.

Threads can be created in two ways:

- There are two ways to create Thread object
  1. Thread class is sub classed and then creating a new object of that class.
  2. Implementation through the Runnable interface.
- Both the cases requires the run method to be implemented.

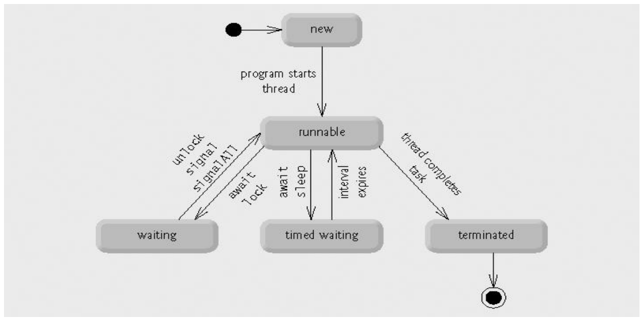


Figure 18: Life Cycle of a thread.

Thread States can be defined as follows:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. **Runnable:** After a newly born thread is started, the thread becomes runnable.
- **Waiting:** A thread might transitions to the waiting state while the thread waits for another thread or to perform I/0.
  Timed waiting: A runnable thread can also execute the timed waiting state for the ceratin interval of time. While in this state a thread can transitions back to the runnable state when specified time interval expirers.
  Terminated: After completion of the task, a runnable thread voluntarily enters the terminated state when it completes its task or otherwise terminates.

7.1.2 CONSISTENCY AND SYNCHRONIZATION

Critical sections of the code can be protected using the synchronization functionality. The *synchronized* keyword in Java ensures:

- Only a single thread can execute a block of code at the same time
- Each thread entering a synchronized block of code sees the effects of all previous modifications that were guarded by the same lock

Synchronization is necessary for mutually exclusive access to blocks of and for reliable communication between threads. We can use the *synchronized* keyword for the definition of a method. This would ensure that only one thread can enter this method at the same time. Another threads which is calling this method would wait until the first threads leaves this method. Th lock can be implemented in the following way:

```
public synchronized void critical ()
{
// We can write
// program here
}
```

Volatile Keyword

If a variable is declared with the *volatile* keyword then it is guaranteed that any thread that reads the field will see the most recently written value. The *volatile* keyword will not perform any mutual exclusive lock on the variable.

7.1.3 FORK JOIN POOL METHOD.

In java the executer service can be implemented through Fork Join and Pool Model.

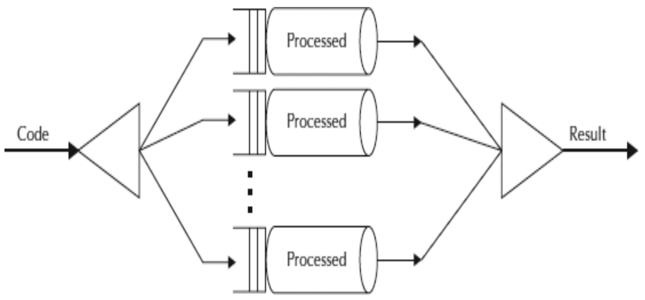


Figure 19:Fork,Join and Pool Model

The Fork Join Pool is basically a specialized implementation of Executor Service. In this we create an instance of Fork Join Pool by providing the target parallelism level i.e. the number of processors.

ForkJoinPool pool = new ForkJoinPool(numberOfProcessors);

WherenumberOfProcessors=Runtime.getRunTime().availableProcessors();

There are three different ways of submitting a task to the ForkJoinPool.

1) execute() method //Desired asynchronous execution; call its fork method to split the work between multiple threads.

2) invoke() method: //Await to obtain the result; call the invoke method on the pool.

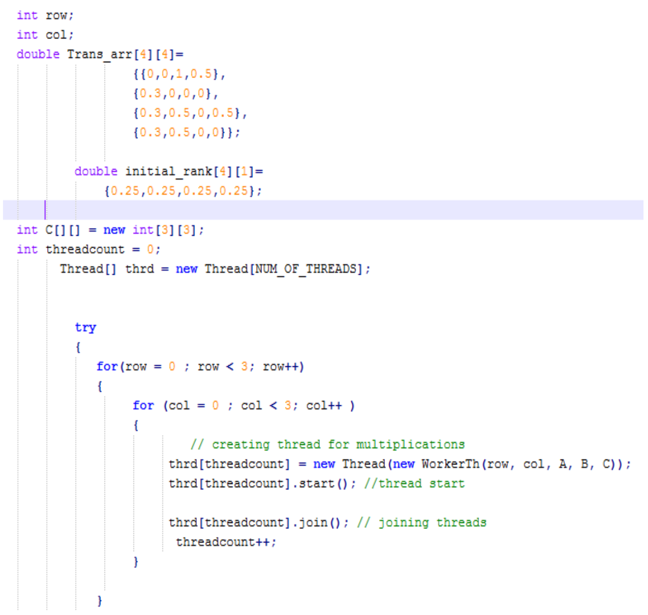
3) submit() method: //Returns a Future object that you can use for checking status and obtaining the result on its completion

7.1.4 JAVA MEMORY MODEL

The java memory model allows the threads to maintain the cached copy of the objects to be used when required by a particular thread to do so. The consistency is monitored through central control. When a thread performs the local modification to the cached objects then the changes are reflected through the central control. All the threads maintain the updates copy of the cached objects.

8. PARALLEL IMPLEMENTATION OF PAGE RANK THROUGH JAVA MULTITHREADING

We have implemented the Page Rank algorithm by following the same approach in java. The Initial Rank vector will contain the initial ranks for the web pages in the web nets. For initial computation we have considered the equal ranks for all the four web pages. The Transitional Vector contains the information about the links in between the web pages. These two matrix are multiplied for some fixed number of iterations after which the system reaches the equilibrium. The matrix multiplication is handled in parallel manner. The threads executes that part of code in parallel which can be implanted and run without any data dependency. Following is the snippet of the code through Java Multithreading.



```
int row;
int col;
double Trans_arr[4][4]=
                {{0,0,1,0.5},
                {0.3,0,0,0},
                {0.3,0.5,0,0.5},
                {0.3,0.5,0,0}};

        double initial_rank[4][1]=
            {0.25,0.25,0.25,0.25};

int C[][] = new int[3][3];
int threadcount = 0;
      Thread[] thrd = new Thread[NUM_OF_THREADS];


        try
        {
           for(row = 0 ; row < 3; row++)
           {
                for (col = 0 ; col < 3; col++ )
                {
                      // creating thread for multiplications
                    thrd[threadcount] = new Thread(new WorkerTh(row, col, A, B, C));
                    thrd[threadcount].start(); //thread start

                    thrd[threadcount].join(); // joining threads
                     threadcount++;
                }

           }
```

Figure 20:Page Rank through Java MultiThreading

9. SUMMARY/CONCLUSION

Our thanks to Dr. Robert Chun (San Jose State University) for helping us throughout our research.

10. REFERENCES

[1] http://www.math.cornell.edu/~mec/Winter2009/RalucaRemus/Lecture3/lecture3.html

[2] http://discover.sjlibrary.org:50080/ebsco-w-b/ehost/pdfviewer/pdfviewer?sid=1495d1a2-57ca-4afd-bec1-ff563b69dd56%40sessionmgr112&vid=4&hid=115

[3] http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1002&amp;context=etd_projects

[4] http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1002&amp;context=etd_projects

[5] http://infolab.stanford.edu/~backrub/google.html

[6] http://biogrid.engr.uconn.edu/REU/Reports_10/Final_Reports/Rifat.pdf