**Title :** Implement a calculator (64 bit Binary Multiplication) application using concurrent lisp.

**Objective :**

Apply concurrent lisp to construct calculator.

**Theory :**

**LISP overview**

John McCarthy invented LISP in 1958, shortly after the development of FORTRAN. It was first implement by Steve Russell on an IBM 704 computer.

It is particularly suitable for Artificial Intelligence programs, as it processes symbolic information effectively.

Common Lisp originated, during the 1980s and 1990s, in an attempt to unify the work of several implementation groups, which were successors to Maclisp like ZetaLisp and NIL (New Implementation of Lisp) etc.

It serves as a common language, which can be easily extended for specific implementation. Programs written in Common LISP do not depend on machine-specific characteristics, such as word length etc.

**Features of Common LISP**

* It is machine-independent
* It uses iterative design methodology, and easy extensibility.
* It allows updating the programs dynamically.
* It provides high level debugging.
* It provides advanced object-oriented programming.
* It provides convenient macro system.
* It provides wide-ranging data types like, objects, structures, lists, vectors, adjustable arrays, hash-tables, and symbols.
* It is expression-based.
* It provides an object-oriented condition system.
* It provides complete I/O library.
* It provides extensive control structures.

**Applications Built in LISP**

Large successful applications built in Lisp.

* Emacs
* G2
* AutoCad
* Igor Engraver
* Yahoo Store

**LISP functions**

This section describes a number of simple operations on lists, i.e., chains of cons cells.

* **cl-caddr** *x*

This function is equivalent to (car (cdr (cdr*x*))). Likewise, this package defines all 24 c*xxx*r functions where *xxx* is up to four ‘a’s and/or ‘d’s. All of these functions are setf-able, and calls to them are expanded inline by the byte-compiler for maximum efficiency.

* **cl-first** *x*

This function is a synonym for (car *x*). Likewise, the functions cl-second, cl-third, ..., through cl-tenth return the given element of the list *x*.

* **cl-rest** *x*

This function is a synonym for (cdr*x*).

* **cl-endp** *x*

Common Lisp defines this function to act like null, but signaling an error if x is neither a nil nor a cons cell. This package simply defines cl-endp as a synonym for null.

* **cl-list-length** *x*

This function returns the length of list *x*, exactly like (length *x*), except that if *x* is a circular list (where the cdr-chain forms a loop rather than terminating with nil), this function returns nil. (The regular length function would get stuck if given a circular list. See also the safe-length function.)

* **cl-list\****arg&rest others*

This function constructs a list of its arguments. The final argument becomes the cdr of the last cell constructed. Thus, (cl-list\* *abc*) is equivalent to (cons *a* (cons *bc*)), and (cl-list\* *ab* nil) is equivalent to (list *ab*).

* **cl-ldiff** *list sublist*

If *sublist* is a sublist of *list*, i.e., is eq to one of the cons cells of *list*, then this function returns a copy of the part of *list* up to but not including *sublist*. For example, (cl-ldiff x (cddr x)) returns the first two elements of the list x. The result is a copy; the original *list* is not modified. If *sublist* is not a sublist of *list*, a copy of the entire *list* is returned.

* **cl-copy-list** *list*

This function returns a copy of the list *list*. It copies dotted lists like (1 2 . 3) correctly.

* **cl-tree-equal** *x y* &key :test :test-not :key

This function compares two trees of cons cells. If *x* and *y* are both cons cells, their cars and cdrs are compared recursively. If neither *x* nor *y* is a cons cell, they are compared by eql, or according to the specified test. The :key function, if specified, is applied to the elements of both trees.

**LISP operators**

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. LISP allows numerous operations on data, supported by various functions, macros and other constructs.

The operations allowed on data could be categorized as:

* Arithmetic Operations
* Comparison Operations
* Logical Operations
* Bitwise Operations

**Arithmetic Operations**

The following table shows all the arithmetic operators supported by LISP.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| + | Adds two operands | (+AB) will give 30 |
| - | Subtracts second operand from the first | (- A B) will give -10 |
| \* | Multiplies both operands | (\* A B) will give 200 |
| / | Divides numerator by de-numerator | (/ B A) will give 2 |
| mod,rem | Modulus Operator and remainder of after an integer division | (mod B A )will give 0 |
| Incf | Increments operator increases integer value by the second argument specified | (incf A 3) will give 13 |
| Decf | Decrements operator decreases integer value by the second argument specified | (decf A 4) will give 9 |

**Comparison Operations**

Following table shows all the relational operators supported by LISP that compares between numbers. However unlike relational operators in other languages, LISP comparison operators may take more than two operands and they work on numbers only.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| = | Checks if the values of the operands are all equal or not, if yes then condition becomes true. | (= A B) is not true. |
| /= | Checks if the values of the operands are all different or not, if values are not equal then condition becomes true. | (/= A B) is true. |
| > | Checks if the values of the operands are monotonically decreasing. | (> A B) is not true. |
| < | Checks if the values of the operands are monotonically increasing. | (< A B) is true. |
| >= | Checks if the value of any left operand is greater than or equal to the value of next right operand, if yes then condition becomes true. | (>= A B) is not true. |
| <= | Checks if the value of any left operand is less than or equal to the value of its right operand, if yes then condition becomes true. | (<= A B) is true. |
| Max | It compares two or more arguments and returns the maximum value. | (max A B) returns 20 |
| Min | It compares two or more arguments and returns the minimum value. | (min A B) returns 20 |

**Logical Operations on Boolean Values**

Common LISP provides three logical operators: **and, or,** and **not** that operates on Boolean values.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| And | It takes any number of arguments. The arguments are evaluated left to right. If all arguments evaluate to non-nil, then the value of the last argument is returned. Otherwise nil is returned. | (and A B) will return NIL. |
| Or | It takes any number of arguments. The arguments are evaluated left to right until one evaluates to non-nil, in such case the argument value is returned, otherwise it returns **nil**. | (or A B) will return 5. |
| Not | It takes one argument and returns **t** if the argument evaluates to **nil.** | (not A) will return T. |

**Mathematical Model:**

**Conclusion :**

Hence in this way we have implemented a calculator (64 bit Binary Multiplication) application using concurrent lisp.

**Title:** Vedic Mathematics method to find square of 2-digit number is used in a distributed programming. Use shared memory and distributed (multi-CPU) programming to complete the task.

**Objective :**

To study of Vedic mathematics to find square of 2-digit number is used in a distributed programming.

**Theory:**

**What is vedic mathematics?**

Vedic Mathematics is the name given to the ancient system of Indian Mathematics which was rediscovered from the Vedas between 1911 and 1918 by Sri BharatiKrsnaTirthaji (1884-1960). According to his research all of mathematics is based on [**sixteen Sutras**](http://www.vedicmaths.org/resources/sutras), or word-formulae. For example, 'Vertically and Crosswise` is one of these Sutras. These formulae describe the way the mind naturally works and are therefore a great help in directing the student to the appropriate method of solution.

Perhaps the most striking feature of the Vedic system is its coherence. Instead of a hotch-potch of unrelated techniques the whole system is beautifully interrelated and unified: the general multiplication method, for example, is easily reversed to allow one-line divisions and the simple squaring method can be reversed to give one-line square roots. And these are all easily understood. This unifying quality is very satisfying, it makes mathematics easy and enjoyable and encourages innovation.

In the Vedic system 'difficult' problems or huge sums can often be solved immediately by the Vedic method. These striking and beautiful methods are just a part of a complete system of athematics which is far more systematic than the modern 'system'. Vedic Mathematics manifests the coherent and unified structure of mathematics and the methods are complementary, direct and easy.

The simplicity of Vedic Mathematics means that calculations can be carried out mentally (though the methods can also be written down). There are many advantages in using a flexible, mental system. Pupils can invent their own methods, they are not limited to the one 'correct' method. This leads to more creative, interested and intelligent pupils.

Interest in the Vedic system is growing in education where mathematics teachers are looking for something better and finding the Vedic system is the answer. Research is being carried out in many areas including the effects of learning Vedic Maths on children; developing new, powerful but easy applications of the Vedic Sutras in geometry, calculus, computing etc. But the real beauty and effectiveness of Vedic Mathematics cannot be fully appreciated without actually practising the system. One can then see that it is perhaps the most refined and efficient mathematical system possible.

[**Sixteen ( 16 ) Sutras of Vedic Mathematics**](http://easyvedicmaths.blogspot.in/2009/04/sixteen-16-sutras-of-vedic-mathematics.html)

|  |  |  |
| --- | --- | --- |
| **Sr. No.** | **Sutras** | **Meaning** |
| 1. | EkadhikinaPurvena | By one more than the previous one |
| 2. | NikhilamNavatashcaramamDashatah | [All from 9 and the last from 10](http://easyvedicmaths.blogspot.com/2010/07/this-is-explanation-to-one-of-16-sutras.html) |
| 3. | Urdhva-Tiryagbyham | Vertically and crosswise |
| 4. | ParaavartyaYojayet | Transpose and adjust |
| 5. | ShunyamSaamyasamuccaye | When the sum is the same that sum is zero. |
| 6. | (Anurupye) Shunyamanyat | If one is in ratio, the other is zero |
| 7. | Sankalana-vyavakalanabhyam | By addition and by subtraction |
| 8. | Puranapuranabyham | By the completion or non-completion |
| 9. | Chalana-Kalanabyham | Differences and Similarities |
| 10. | Yaavadunam | Whatever the extent of its deficiency |
| 11. | Vyashtisamanstih | Part and Whole |
| 12. | ShesanyankenaCharamena | The remainders by the last digit |
| 13. | Sopaantyadvayamantyam | The ultimate and twice the penultimate |
| 14. | EkanyunenaPurvena | By one less than the previous one |
| 15. | Gunitasamuchyah | The product of the sum is equal to the sum of the product |
| 16. | Gunakasamuchyah | The factors of the sum is equal to the sum of the factors |

**1. EkadhikenaPurvena**

The Sutra (formula) EkadhikenaPūrvena means: “By one more than the previous one”.

* Squares of numbers ending in 5 :

Now we relate the sutra to the ‘squaring of numbers ending in 5’. Consider the example 252

Here the number is 25. We have to find out the square of the number. For the number 25, the last digit is 5 and the 'previous' digit is 2. Hence, 'one more than the previous one', that is, 2+1=3. The Sutra, in this context, gives the procedure to multiply the previous digit 2 by one more than itself, that is, by 3. It becomes the L.H.S (left hand side) of the result, that is, 2 X 3 = 6. The R.H.S (right hand side) of the result is 52, that is 25. Thus 252= 2 X 3 / 25 = 625. In the same way,

352= 3 X (3+1) /25 = 3 X 4/ 25 = 1225;

652= 6 X 7 / 25 = 4225;

1052= 10 X 11/25 = 11025;

1352= 13 X 14/25 = 18225;

**Share Memory:**

In computer programming, shared memory is a method by which program [process](http://whatis.techtarget.com/definition/process)es can exchange data more quickly than by reading and writing using the regular operating system services. For example, a [client](http://searchenterprisedesktop.techtarget.com/definition/client) process may have data to pass to a [server](http://whatis.techtarget.com/definition/server) process that the server process is to modify and return to the client. Ordinarily, this would require the client writing to an output file (using the [buffer](http://searchcio-midmarket.techtarget.com/definition/buffer)s of the operating system) and the server then reading that file as input from the buffers to its own work space. Using a designated area of shared memory, the data can be made directly accessible to both processes without having to use the system services. To put the data in shared memory, the client gets access to shared memory after checking a [semaphore](http://searchenterpriselinux.techtarget.com/definition/semaphore) value, writes the data, and then resets the semaphore to signal to the server (which periodically checks shared memory for possible input) that data is waiting. In turn, the server process writes data back to the shared memory area, using the semaphore to indicate that data is ready to be read.

**Distributed Memory:**

Distributed memory refers to a [multiple-processor computer system](http://en.wikipedia.org/wiki/Multiprocessing) in which each [processor](http://en.wikipedia.org/wiki/Central_processing_unit) has its own private [memory](http://en.wikipedia.org/wiki/Computer_memory). Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors. In contrast, a [shared memory](http://en.wikipedia.org/wiki/Shared_memory) multi processor offers a single memory space used by all processors. Processors do not have to be aware where data resides, except that there may be performance penalties, and that race conditions are to be avoided.

**Mathematical Model:**

**Conclusion :**

Hence we have implemented Vedic mathematics to find square of 2-digit number is used in a distributed programming. Use shared memory and distributed (multi-CPU) programming to complete the task.

**Title :** Implement a Parallel ODD-Even Sort algorithm using GPU or ARM equivalent.

**Objective :**

To study Parallel ODD-Even Sort algorithm using GPU or ARM equivalent.

**Theory :**

Parallelism on chip level is the hub for advancements in micro processor architectures for high performance computing. The core-processors, in personal computers, were not sufficient for high data- computation intensive tasks.

As a result modular and specialized hardware in the form of sound cards or graphic accelerators are increasingly present in most personal computers.

Graphics cards or graphics processing units (GPU), introduced primarily for high-end gaming requiring high resolution.

The GPU itself is a multi-core processor having support for thousands of threads running concurrently. GPU's are result of dozens of streaming processors with hundreds of core aligned in a particular way forming a single hardware unit.

Performance evaluation in GFLOPS (Giga Floating Point Operations per Second) shows that GPU's outperforms their CPU counterparts. For example: a high-end Core I7 processor (3.46 GHz) delivers up to a peak of 55.36 GFLOPs1.

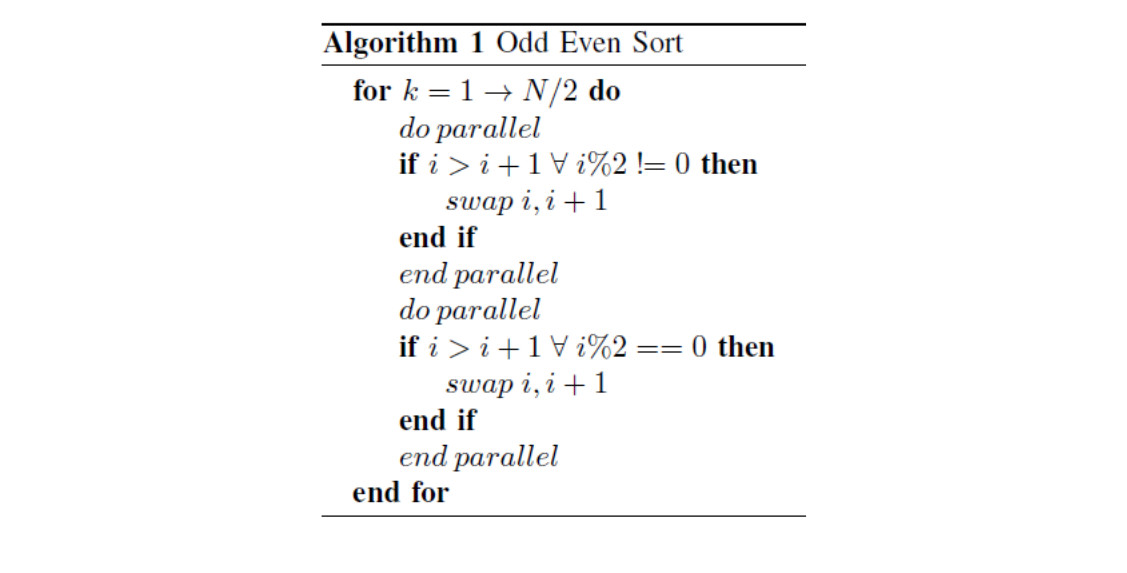
**Parallel sorting algorithms**

Sorting on GPU require transferring data from main memory to on-board GPU global memory. Although on-device bandwidth is in the range of 144Gb/s, thus only those sorting techniques are efficient which require minimum amount of synchronization because the PCI bandwidth is to the range of 2.5Gb/s. i.e., synchronization and memory transfers between CPU and GPU will affect system performance adversely.

Compared to serial sorting algorithms, parallel algorithms are designed requiring high data independence between various elements for achieving better performance. Those techniques which involve large data dependency are categorized as sequential sorting algorithms.

**Odd-Even Sort**

The odd-even sort is a parallel sorting algorithm and is based on bubble-sort technique. Adjacent pairs of items in an array are exchanged if they are found to be out of order. What makes the technique distinct from bubble-sort is the technique of working on disjointed pairs, i.e., by using alternating pairs of odd-even and even-odd elements of the array. The technique works in multiple passes on a queue Q of size N. In each pass, elements at odd-numbered positions perform a comparison check based on bubble-sort, after which elements at even- numbered positions do the same. The maximum number of iterations or passes for odd-even sort s N2. Total running time for this technique is Ο(log 2 N ) . The algorithm works as:-

****

**Mathematical Model:**

**Conclusion :**

Hence we have implemented a Parallel ODD-Even Sort algorithm using GPU or ARM equivalent.

**Title :** Implement *nxn* matrix parallel multiplication using CUDA/OpenCL GPU, use shared memory.

**Objective :**

To study *nxn* matrix parallel multiplication using CUDA/OpenCL GPU, use shared memory.

**Theory :**

CUDA

* It is an extension to the C language that allows GPU code to be written in regular C. The code is either written for the host processor (the CPU) or to the device processor (the GPU).
* The host processor spawns multithread tasks onto the GPU device. The GPU has its own internal scheduler that will then allocate the kernels to avialble GPU hardware.
* A big problem, what fraction of the code can be run in parallel.
* The maximum speedup possible is limited by the quantity of serial code. If you have an infinite amount of processing power and could do the parallel tasks in zero time, you would still be left with the time from the serial code part. NVIDIA is committed to providing support to CUDA.

**Cuda Installation Windows.**

1. System Requirements

Hardware Requirments: Cuda enabled Graphics Card

(Check wheater your System Comes with graphics card)

OS: Windows 7/8

Visual Studio 12 Ultimate Edition

Cuda 6.5 toolkit

2. Install Visual Studio 12 Ultimate edition

(Select Option Visual C++)

3. Install Cuda 6.5 toolkit

4. Launch Visual Studio

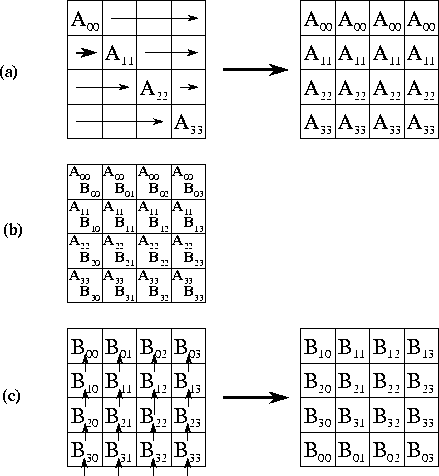
5. \*(first time only) Select Visual C++

6. File -> New Project -> NVIDIA -> Cuda 6.5 (\*Ver) -> Enter Name for project -> OK -> Edit .cu file -> run local debuger.

**Matrix Parellel Multiplication:**

In [mathematics](http://en.wikipedia.org/wiki/Mathematics), **matrix multiplication** is a [binary operation](http://en.wikipedia.org/wiki/Binary_operation) that takes a pair of [matrices](http://en.wikipedia.org/wiki/Matrix_(mathematics)), and produces another matrix. [Numbers](http://en.wikipedia.org/wiki/Number) such as the [real](http://en.wikipedia.org/wiki/Real_number) or [complex numbers](http://en.wikipedia.org/wiki/Complex_number) can be [multiplied](http://en.wikipedia.org/wiki/Multiplication) according to [elementary arithmetic](http://en.wikipedia.org/wiki/Elementary_arithmetic). On the other hand, matrices are *arrays of numbers*, so there is no unique way to define "the" multiplication of matrices. As such, in general the term "matrix multiplication" refers to a number of different ways to multiply matrices. The key features of any matrix multiplication include: the number of rows and columns the original matrices have and specifying how the entries of the matrices generate the new matrix.

Because of the nature of matrix operations and the layout of matrices in memory, it is typically possible to gain substantial performance gains through use of [parallelization](http://en.wikipedia.org/wiki/Parallelization) and [vectorization](http://en.wikipedia.org/wiki/Automatic_vectorization). Several algorithms are possible, among which [divide and conquer](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithms) algorithms based on the [block matrix](http://en.wikipedia.org/wiki/Block_matrix) decomposition



P= n ***Tpar = O(n2)***

Each instance of inner loop is independent and can be done by a separate processor.

**Cost optimal** since O(n3) = n \* O(n2)

P = n2 ***Tpar = O(n)***

One element of C (cij) is assigned to each processor.

**Cost optimal** since O(n3) = n2 x O(n)

P = n3 ***Tpar = O(*log *n)***

n processors compute one element of C (cij) in parallel (O(log n))

**Not cost optimal** since O(n3) < n3 \* O(log n)

***O(*log *n)*** lower bound for parallel matrix multiplication.

We implement a fast matrix multiplication algorithm with asymptotic complexity O(N2.775) for square N×N matrices. In terms of asymptotic complexity, this is the fastest matrix multiplication algorithm implementation to date. However, our performance results show that this algorithm is not practical for the problem sizes that we consider. Overall, we find that Strassen’s algorithm

**Mathematical Model:**

**Conclusion :**

Hence we have implemented *nxn* matrix parallel multiplication using CUDA/OpenCL GPU, use shared memory.

**Title :** Implement a Multi-threading application for echo server using socket programming in JAVA.

**Objective :**

To study a Multi-threading application for echo server using socket programming in JAVA.

**Theory :**

**Socket Programming :**

Socket programming is useful for building client-server applications.

**The server :**

Creates a socket with some port number (>1023):   
*ServerSocketechoServer = new ServerSocket(6789);*

Waits for client connection :   
*Socket clientSocket = echoServer.accept();*

Gets input/output streams :   
*BufferedReader is = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));*  
*PrintStreamos = new PrintStream(clientSocket.getOutputStream());*

Exchanges information with the client :   
*String line = is.readLine();*  
*os.println( "Echo: " + line );*

Clean up :   
*is.close();   os.close();*  
*clientSocket.close();   echoServer.close();*

**The client :**

Creates a socket with the same port number :   
*Socket clientSocket = new Socket( "localhost", 6789 );*

Gets input/output streams.

Exchanges information with the server.

Clean up.

**ECHO Server :**

**What is it and what is it for ?**

An "echo server" is a server that does nothing more than sending back whatever is sent to it. Hence the name : echo What can you use it for ? Whatever you feel like. Practical applications could be network and connectivity testing and troubleshooting. Assume you've build a rather complex network with VLANs and subnets, and really strick firewalls between those subnets, and you're beginning to wonder if a client on one segment of the network will still be able to connect to your web server, database server, ... on some other segment. A ping or a traceroute will establish if the server (IP address) can be reached but does not tell you if an application will be able to connect to the desired port on the server and whether a reply from the server will be able to reach the client again.

This "echo server" can be set up to listen on any desired (tcp) port to simulate whatever application you want to run (eg web server = port 80, Microsoft SQL Server = port 1433, etc). From the client machine, you can then telnet to this port. When a telnet connection has been established, everything you type will be echoed back to your screen, indicating that the telnet client and the echo server can talk to each other : you've established connectivity at the application level.

In a similar way, you can use this echo server to troubleshoot networks, test a firewall (eg "if I have a server listening on port 123, wil my firewall allow connections to it ?) and so on.

**Echo server :**

1. The client reads a line from its standard input and writes that line to the server.

2. The server reads a line from its network input and echoes the line back to the client over the network.

3. The client reads the echoed line from the network and prints it on its standard output.

**Multithreaded Server Advantages :**

The advantages of a multithreaded server compared to a single threaded server are summed up below:

Less time is spent outside the accept() call.

Long running client requests do not block the whole server.

In a single threaded server long running requests may make the server unresponsive for a long period. This is not true for a multithreaded server, unless the long-running request takes up all CPU time and/or network bandwidth.

**Mathematical Model:**

**Conclusion :**

Hence we have implemented a Multi-threading application for echo server using socket programming in JAVA.

**Title :** Implement a Parallel Quick Sort algorithm using NVIDIA GPU or equivalent ARM board.

**Objective :**

To study Parallel Quick Sort algorithm using NVIDIA GPU or equivalent ARM board.

**Theory :**

**Features**

* Similar to mergesort - divide-and-conquer recursive algorithm
* One of the fastest sorting algorithms
* Average running time O(NlogN)
* Worst-case running time O(N2)

**Basic idea**

1. Pick one element in the array, which will be the *pivot*.
2. Make one pass through the array, called a *partition* step, re-arranging the entries so that:
   * + the pivot is in its proper place.
     + entries smaller than the pivot are to the left of the pivot.
     + entries larger than the pivot are to its right.
3. Recursively apply quicksort to the part of the array that is to the left of the pivot,   
   and to the right part of the array.

Here we don't have the merge step, at the end all the elements are in the proper order.

**Algorithm**

**STEP 1**. Choosing the pivot

Choosing the pivot is an essential step.   
Depending on the pivot the algorithm may run very fast, or in quadric time.:

* 1. Some fixed element: e.g. the first, the last, the one in the middle

This is a bad choice - the pivot may turn to be the smallest or the largest element,   
then one of the partitions will be empty.

* 1. Randomly chosen (by random generator ) - still a bad choice.
  2. The median of the array (if the array has N numbers, the median is the [N/2] largest number. This is difficult to compute - increases the complexity.
  3. The median-of-three choice: take the first, the last and the middle element.   
     Choose the median of these three elements.

Example:

8, 3, 25, 6, 10, 17, 1, 2, 18, 5

The first element is 8, the middle - 10, the last - 5.  
The median of [8, 10, 5] is 8

**STEP 2**. Partitioning

Partitioning is illustrated on the above example.

1. The first action is to get the pivot out of the way - swap it with the last element

5, 3, 25, 6, 10, 17, 1, 2, 18, 8

2. We want larger elements to go to the right and smaller elements to go to the left.

Two "fingers" are used to scan the elements from left to right and from right to left:

[5, 3, 25, 6, 10, 17, 1, 2, 18, 8]

^ ^

i j

* While i is to the left of j, we move i right, skipping all the elements   
  less than the pivot. If an element is found greater then the pivot, i stops
* While j is to the right of i, we move j left, skipping all the elements   
  greater than the pivot. If an element is found less then the pivot, j stops
* When both i and j have stopped, the elements are swapped.
* When i and j have crossed, no swap is performed, scanning stops,   
  and the element pointed to by i is swapped with the pivot .

In the example the first swapping will be between 25 and 2, the second between 10 and 1.

3. Restore the pivot.

After restoring the pivot we obtain the following partitioning into three groups:

[5, 3, 2, 6, 1] [ 8 ] [10, 25, 18, 17]

**STEP 3**. Recursively quicksort the left and the right parts

**Mathematical Model:**

**Conclusion :** Hence we have implemented a Parallel Quick Sort algorithm using NVIDIA GPU

**Title:** Implement n-ary search algorithm using OPENMP.

**Prerequisites (HW/SW Requirement):** Fedora 20, GCC, OPENMP Compiler Environment.

**Theory:**

**Introduction to N-ary Search:**

* If we have N threads available, we can develop a concurrent N-ary search. This search identifies N well-spaced points within the search array bounds and compares the key of the corresponding records to the search key.
* Each thread does one of the N comparisons. There are three possible outcomes from these comparisons. The first is that the item of interest is found and the search is complete; the second is that the item key examined is less than the search key; the third is that the item key examined is greater than the search key. If no search key match is found, a new, smaller search array is defined by the two consecutive index points whose record keys were found to be less than the search key and greater than the search key.
* The N-ary search is then performed on this refined search array. As with the serial version of binary search, the process is repeated until a match is found or the number of items in the search array is zero. A pictorial description of this algorithm is shown in Figure1.

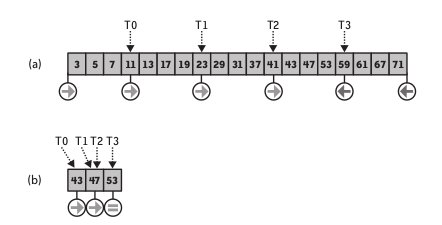


FIGURE 1. N-ary search example with four threads

* Given the sorted array of prime numbers in Figure (a), let’s say we want to determine
* whether the value 53 is in the array and where it can be found. If there are four threads (T0 to T3), each computes an index into the array and compares the key value found there to the search key.
* Threads T0, T1, and T2 all find that the key value at the examined position is less than the search key value. Thus, an item with the matching key value must lie somewhere to

the right of each of these thread’s current search positions (indicated by the circle arrows).

* Thread T3 determines that the examined key value is greater than the search key, and the

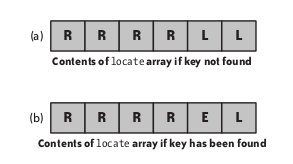
matching key can be found to the left of this thread’s search position (left-pointing circled

arrow).

* Notice the circled arrows at each end of the array. These are attached to “phantom” elements just outside the array bounds.
* The results of the individual key tests define the subarray that is to become the new search array. Where we find two consecutive test results with opposite outcomes, the corresponding indexes will be just outside of the lower and upper bounds of the new search array.
* Figure (a) shows that the test results from threads T2 (less than search key) and T3 (greater than search key) are opposite. The new search array is the array elements between the elements tested by these two threads. Figure (b) shows this subarray and the index positions that are tested by each thread. The figure shows that during this second test of element key values, thread T3 has found the element that matches the search key (equals sign in circle).
* Consider the case where we want to find a composite value, like 52, in a list of prime numbers.
* The individual key results by the four threads shown in Figure (a) would be the same.
* The subarray shown in Figure (b) would have the same results, except that the test by T3 would find that the key value in the assigned position was greater than the search key (and the equals sign would be a left-pointing arrow).
* The next round of key comparisons by threads would be from a subarray with no elements, bounded by the array slots holding the key values of 47 and 53. When threads are confronted with the search of an empty search space, they know that the key is not to be found.
* This is obviously more complex than a simple binary search. The algorithm must coordinate the choices of index positions that each thread needs to test, keep and store the results of each test such that multiple threads can examine those results, and compute the new search array bounds for the next round of key tests. From this quick description, it’s clear that we’ll need some globally accessible data and, more importantly, we need a barrier between the completion of the key tests and the examination of the results of those tests.

**But First, a Serial Version**

* Though it doesn’t make much sense to implement the N-ary search algorithm in serial, we’re going to do it anyway. By examining the serial version first, we can identify all the pieces and parts of the algorithm, which is an adaptation of the CREW SEARCH algorithm from Selim Akl’s The Design and Analysis of Parallel Algorithms.
* Once you’ve got that down, creating a concurrent version using OpenMP will be very straightforward. Example 1. contains a serial version of N-ary search.
* The parameters to the NarySearch() function are the array to be searched (A), the initial index bounds of the search space (lo and hi), the value of the key the algorithm is attempting to find (key), the number of intervals to divide the search array into each round (Ntrval), and the pointer to return the index position of a record with a matching key (pos).
* The code assumes that the pos value has been initialized to −1 before calling this function. The Ntrvl parameter is likely going to be the number of threads; I will explain why we’re implementing the function with this parameter when I discuss the concurrent solution.
* This function declares two arrays. These will hold the index of the key test element (mid) and the results of the corresponding key comparison (locate). I’ve chosen to use a character array for the latter to denote whether the element’s key is too big, too small, or just right. The lettersindicate that the desired element with the matching search key will be found to the right (R) or to the left (L), or that the search and item keys are exactly equal (E).
* Any three values can be used here, as long as two consecutive elements, when compared, are found to have either the same value or different values. The locate array has two more slots than the number of intervals to be used. The first and last elements mimic the key comparison results from the phantom array slots just outside the bounds of the search array and are initialized to R and L, respectively.
* The while loop continues to iterate a new search round as long as there is at least one array item to be searched and no element of the array has been found to have a matching key. The mid[0] element holds the index of the leftmost bound of the search array, and step is the number of slots between each element to be tested in the current round. Note that the code declares step as a float rather than an integer. By computing indexes with floating-point numbers and truncating any fractional parts before utilizing the value, we can ensure that the computed key test indexes are more equitably spread out within the search.
* In addition, this code includes the lmid variable to give threads a local value for holding the index value of a key to be compared against the search key. This addition follows Simple Rule 7.
* The body of the while loop has three separate parts. The first for loop computes an offset into the search array for each interval and stores the computed index into the corresponding element of mid (if the computed index is outside of the search array, the saved value is hi + 1 and this will be treated like the rightmost phantom element). Next, the key found at each of these positions is compared to the search key and the relationship of the location of the search key to the element key is recorded in the corresponding locate element.
* If there is a match on the search key, the value of pos is updated with the index of the matching element. Once all of the test indexes are computed and the results of the key comparisons have been noted, the second for loop checks all consecutive pairs of elements from the locate array to see whether the characters stored there are different. As shown in Figure 1, this difference indicates that the record that matches the search key will lie between the two corresponding index points in the search array. When such a difference is found, the values of lo and hi are updated to set the bounds of the search array for the next iteration of the while loop.
* The third part of the while loop body does this same locate element test, but it does so with the rightmost phantom index (Ntrvl + 1). If this value is different from the locate element of the final computed test index, then only the lower bound of the search array needs to be updated.
* Could there be more than one difference in the locate array? If the search key doesn’t match any of the keys at the test index positions, the locate array will have only R and L characters.
* This is shown in Figure 2 (a). In fact, there will be one or more Rs followed by one or more Ls. Because the keys in the search array are sorted, all positions in locate to the left of the rightmost R must also be Rs, and all positions in locate to the right of the leftmost L must be Ls.
* If the search key is found, though, there will be two differences in the locate array, as shown in Figure 2 (b). While this will cause the serial algorithm to update the lo and hi bounds twice, when the algorithm expects this to be done only once, the next while conditional expression evaluation will yield FALSE. Thus, this extra bounds update will simply be discarded as the NarySearch() function returns from a successful search.



**FIGURE 2. Example of the locate array contents for two different cases**

**At Last, the Concurrent Solution**

* Example 2 contains the OpenMP version of N-ary search. The main reason that we’re using OpenMP for the concurrent version of NarySearch() is because of the need for two barrier synchronizations.
* We need the first between the determination of the location of the matching element from the data array and the search through the locate array to identify the difference between two consecutive items. We need the second barrier between the update of lo and hi and the start of the next iteration of the while loop.
* Of course, the decision to use OpenMP is helped along by the fact that the algorithm has a data decomposition solution and uses for loops for iteration within the while loop. The additions to the code from Example 1. Have been highlighted in bold.
* The parallel region encloses the while loop, which means that each thread on the team will be executing all iterations of the while loop (the curly braces on the parallel pragma aren’t strictly needed, but they serve to point out the extent of the parallel region).
* For threads to keep in sync with these iterations, the variables for lo, hi, and pos must be globally accessible to each thread. This will ensure that the results of the while conditional test are the same in all threads and that the threads will end the parallel region at the same time.
* All threads share the step variable before the OpenMP loop worksharing construct begins, but the initial assignment of step and mid[0] must be done in serial. Th firstprivate clause will create a private copy of this variable for each thread and initialize that copy with the value computed outside of the worksharing construct. Having a local copy of step is useful if there are fewer threads than intervals to check by avoiding multiple accesses of the global step by each thread.
* The computation of the test key indexes and the discovery of the correlation of key values found at those points are all independent. Thus, the loop worksharing construct will divide up the intervals (via division of the loop iterations) among the threads of the team.
* If the number of intervals (Ntrvl) is equal to the number of threads, we will get one interval per thread; if there are more intervals than threads, multiple intervals will be assigned to threads. By including the interval parameter (Ntrvl), we can better control the granularity of computationsn assigned to each thread.
* You may find that the overhead of doing only one interval per thread is too high versus the amount of computation. So if you specify, say, 10 intervals per thread, there is relatively less OpenMP overhead per computation per iteration of the outer while loop per thread.
* In addition, you execute fewer iterations of the while loop with 80 intervals than with 8. This assumes that you have a data set large enough to benefit from probing 80 locations.
* Using the single worksharing construct makes certain that the second for loop and the last

boundary check of the algorithm will be executed in serial. We’re using single rather than

master for the implicit barrier to ensure that values for lo and hi are updated before

thread undertake the next iteration of the while loop.

* Actually, only the last boundary check is a serial operation. We could put the second for loop in a loop worksharing construct, especially if we have more intervals than threads. However, for a small number of intervals, there will be less memory traffic into cache lines holding the locate array if one thread alone does the check.
* Since only one interval test will find a difference in consecutive locate elements, only one

thread will update the lo and hi values. As a result, even if we ran the second for loop

concurrently, there would be no need for mutually exclusive access and no need for any

synchronization objects.

* This is true even when a matching element from the array is found and there are two differences in locate. Yes, this could lead to the situation of a race to store conflicting values into lo and hi, but the while condition test immediately after will halt the

search, since the item of interest has been found.

**Steps for Execution:**

**1. compilation**

gcc -o omp\_naryc -fopenmp omp\_nary.c -lstdc++

export OMP\_NUM\_THREADS=2

**2. run**

./naryc

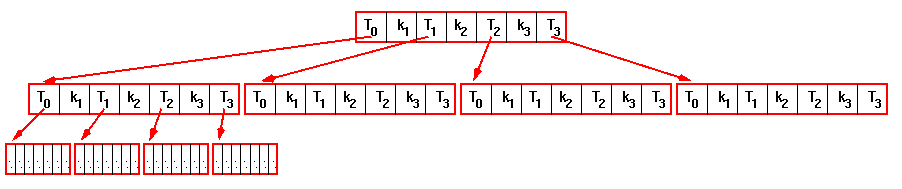
**Input:** array of elements

**Output:** finds the position of a specified input value

**Algorithm:**

If we relax the restriction that each node can have only one key, we can reduce the height of the tree.

|  |  |
| --- | --- |
| An **m-way search tree**   1. is empty or 2. consists of a root containing **j** (1<=**j**<**m**) keys, **kj**, and a set of sub-trees, **Ti**, (**i** = 0..**j**), such that    1. if **k** is a key in **T0**, then **k** <= **k1**    2. if **k** is a key in **Ti** (0<**i**<**j**), then **ki** <= **k** <= **ki+1**    3. if **k** is a key in **Tj**, then **k** > **kj** and    4. all **Ti** are nonempty m-way search trees or all **Ti** are empty | Or in plain English ..   1. A node generally has **m-1** keys and **m** children. Each node has alternating sub-tree pointers and keys: sub-tree | key | sub-tree | key | ... | key | sub\_tree    1. All keys in a sub-tree to the left of a key are smaller than it.    2. All keys in the node *between* two keys are between those two keys.    3. All keys in a sub-tree to the right of a key are greater than it.    4. This is the "standard" recursive part of the definition. |



The height of a complete m-ary tree with **n** nodes is ceiling(**logmn**).

A **B-tree of order m** is an m-way tree in which

1. all leaves are on the same level and
2. all nodes except for the root and the leaves have at least **m**/2 children and at most **m** children. The root has at least 2 children and at most **m** children.

A variation of the B-tree, known as a **B+-tree** considers all the keys in nodes except the leaves as dummies. All keys are duplicated in the leaves. This has the advantage that is all the leaves are linked together sequentially, the entire tree may be scanned without visiting the higher nodes at all.

**Conclusion:**

Hence, we have successfully studied concurrent N-ary search algorithm using OpenMp.

**Title:** Implement concurrent Prim’s algorithm using OPENMP

**Objectives:** To learn multi-core and concurrent programming using OPENMP for Prim’s algorithm

**Aim:** To learn difference between serial and parallel execution of Prim’s algorithm using OPENMP

**Prerequisites (HW/SW Requirement):**

Fedora 20, GCC, OPENMP Compiler Environment

**Theory:**

Prim’s Algorithm takes the almost opposite tactic to Kruskal’s. This algorithm grows the MST by adding the edge that is the minimal distance from a node that is already part of the partial spanning tree. An initial node is chosen as the root of the spanning tree. For all nodes that are not currently in the partial spanning tree, each edge from the node to a node in the tree is considered, and the edge with the minimum weight is nominated. From all the nominated edges, the one with the smallest weight (ties are handled as desired) is added to the tree, which adds a new node to the partial spanning tree. The nominating and choosing process for the smallest weight edge linking a node not present in the partial spanning tree is repeated until all nodes have been added. Following Figure demonstrates the order of edges added to the partial spanning tree under Prim’s Algorithm on the graph.



Figure B6.1 Undirected weighted graph and associated weight matrix



Figure B6.2 Example of Prim’s algorithm

Node A is arbitrarily chosen as the root of the spanning tree in Figure (b). After that, the edge with the lowest weight and that is adjacent to a node in the partial spanning tree is added to the tree. The final results are the same as that of Kruskal’s algorithm. The order of the nodes added, however, is different. For example, the edge (D,B) is included first in Kruskals Algorithm, but is added last in Prim’s Algorithm (for the given graph and starting node). The order of nodes added to the MST by Prim’s Algorithm all depends on which node is chosen as the root.

Besides the order of edge inclusion, there is one other major difference between the two algorithms. Kruskal’s Algorithm needs to sort the edges by length, whereas Prim’s Algorithm simply needs to keep track of the edge that has minimum distance from an unused node to some node in the partial tree. You can use a heap data structure to make the overall smallest length edge available for Kruskal’s Algorithm; a pair of arrays, with an element for each node in the graph, will keep track of which is the minimal edge from each unused node and the length of that edge for Prim’s Algorithm. A simple search of the arrays within Prim’s Algorithm will determine the next node to be added to the tree.

The Prims() function in Example assumes that there are no negative edges in the graph. If negative edges are possible, we will need to modify some of the conditional expressions to account for this. Three parameters are used: the weight matrix for the graph (W), a pointer that will return the minimum spanning tree as a vector of edges (T), and the number of nodes in the graph (N). The declarations in the function include an array to note the nearest node in the partial tree for each other node (nearNode), an array that holds the (minimum) distance from each node to the nearest node in the partial tree (minDist), and a variable to hold the overall minimum from minDist (min).

A node is chosen as the root of the partial minimum spanning tree. Just for convenience, the algorithm chooses node [0]. The initial values for elements of nearNode are set to [0], the only node in the partial tree, and the initial values for minDist will be the corresponding weights from each node to node [0] taken from the weight matrix.

The second for loop in the function iterates N−1 times. Each iteration will add an edge to the partial tree (since a tree with N nodes has N−1 edges). The code takes three steps to find the next edge to add to the tree. First, it finds the node that is the shortest distance from any node in the partial tree. Next, the associated edge is added to the partial tree. Finally, the minimum distances from the nontree nodes to the partial tree are updated in light of the newly added node.

To find the node that is the minimum distance to any node in the partial tree, the standard minimum-seeking algorithm is used. That is, the code in the first j loop examines all nontree elements of minDist in order to find the smallest value stored there (min) and the corresponding node (k) from the graph. Once the minimal distance is found, the corresponding edge is stored in the tree by saving the two nodes defining the edge (nearNode[k], k) into the tree array, T.

Note that the conditional expression used to find the minimum value includes a test to determine whether or not the node has already been added to the tree by first checking to ensure that the minDist entry for a node is nonnegative. This pretest condition is used because the element of minDist that is found to be minimal is overwritten by −1 after the new tree edge is saved. This value simply needs to be something smaller than the smallest edge weight in the graph. Thus, if we use negative edge weights, we might use a value like FLT\_MIN instead.

The third step updates the minimum distance of each nonroot node from a node in the partial tree by checking to see whether the newly added tree node is closer to these nodes. The assignment of −1 to the minDist entries in the previous step will ensure that the nodes already within the partial tree do not get these updates.

**Concurrent Version of Prim’s Algorithm**

Edges need to be added to the partial tree one at a time to ensure that the correct edges are used. We can’t just stick an OpenMP loop worksharing construct around the second i loop of the serial code in Example. So, this leaves us with the mission of trying to parallelize the steps within each iteration of the i loop. Let’s look at these steps, starting from the last and moving to the first. The final step, updating the minimum distances of nontree nodes to the partial tree, is made up of independent operations. Each node is looking at the weight of an edge from itself to the newly added tree node, and updating the corresponding element of minDist. We could use an OpenMP loop worksharing construct on this loop.

The second step, adding the chosen edge to the partial tree, must be done sequentially. If we’re using OpenMP, we could do this with the single construct (the master construct doesn’t have the implicit barrier we need to ensure that the tree and node updates are done before the third step commences). Since each thread will be writing the same values, we could allow all threads to update the same elements of T and minDist. There are no correctness issues for this benign data race, but there would be performance implications of multiple cores updating entries in the same cache line. To decide which method is better, we need to weigh the overhead of the single cache line being moved between cores versus the overhead of pausing threads, running code in serial, and then starting threads up again for the final step. (I’m assuming that the first step can utilize threads, which is a safe bet since I’ve looked ahead and know how it all turns out.)

For the first step, I presume the code can run concurrently. If this is possible, we’d need to have both min and k globally accessible, and that would mean possible synchronization at every iteration of the j loop. That’s just a bit too much overhead. Rather than looking at the bits and pieces used for the computation, we should step back and look at what kind of computation is being done. Lo and behold! This is a reduction operation to find the minimum and, more importantly, the index of that minimum, which is the node of the graph that should next be added to the partial tree.

After recognizing that this first step is just a reduction, you may realize that we can’t do this operation with the OpenMP reduction clause. The algorithm doesn’t need the minimal value stored in minDist, but the index of the element storing that value. Even if we had a min operator for use in the reduction clause (which is supported in FORTRAN), there’s no operator that will return the index of a minimal element. Thus, we will need to write the reduction by hand or use the parallel\_reduce algorithm from TBB. Since this computation is right out of the TBB tutorial examples, let’s go with TBB. Example B56.2 has an adapted version of the tutorial example for the concurrent MST algorithm

as execution of the code progresses? Well, actually, the number of nodes to be seriously considered each time through the loop decreases at a steady pace. As the number of nodes in the partial tree increases, the number of possible nodes that would have associated values updated in the first or third steps will decline. Threads will need to iterate through all nodes assigned by the scheduling algorithms, but fewer and fewer will generate even the chance of some computation in the first and third steps.

In an extreme arrangement, imagine nodes that are assigned to threads in a static fashion and added to the partial spanning tree in indexed order. In the midst of this situation we would have threads without any potential new tree nodes, while other threads would have all the potential nodes. Near the end, then, only one thread would even have the possibility of doing worthwhile computations and updates in the first and third steps.

The algorithm inherently contains a steady reduction in scalability. Does this decline in scalability and the load imbalance posited in the last paragraph have a noticeably adverse affect on the performance? We could alleviate the extreme cases of load imbalance by using a more dynamic scheduling of nodes to threads in each iteration of the i loop. (The TBB task scheduler should yield some modicum of dynamic scheduling automatically, so a schedule clause on the OpenMP pragma should be sufficient.) Even so, there are still scenarios that we could devise to limit the efficacy of dynamic iteration scheduling.

The cause of these imbalances is the need to look through all nodes, tree or nontree, in the first and last steps of the algorithm loop. Would a more dynamic method for keeping only nontree nodes around in the first and third steps be sensible? You might have a better idea, but the first thing I think of is to use a linked list of struct nodes containing the node number, the node in the partial tree that is closest to the node, and the length of the shortest edge between these nodes. The first step goes through all the nodes in the list to find the one that is the minimum distance from the partial tree; the second step adds the edge to the tree and removes the node from the linked list; and the third step updates the distances of the nodes remaining on the list (one way to process linked list nodes concurrently is to create a task to process each node and have those tasks placed in a queue to be accessed by threads—both OpenMP and TBB support this kind of concurrent execution).

Clearly, we would need to change the code in all steps if we adopted the linked list idea. I hope you agree that processing nodes in a linked list is going to require code that is less simple than

running through an array. More importantly, will there be an adequate performance benefit possible by taking the time and trouble to implement such a solution? Again, this is something

that we would need to implement and compare to the original code version for a definitive answer. With the linked list structure, we get good load balance, but we will have a very real reduction of work after each iteration of the outermost loop. For a fixed number of threads, the amount of work will eventually get to the point where each iteration is dwarfed by the threading overhead. At least with the array implementation, threads consider every assigned node, even if it is only to find that the node was already part of the tree.

There may be a crossover point in graph sizes that would favor one implementation over another. I suspect that for large graphs (e.g., thousands or tens of thousands of nodes), it is more likely that dealing with only active, nontree nodes will reap adequate performance and scalability benefits versus the original code in Example B6.3. Such a crossover point, if it exists, would need to be determined experimentally

**Steps for Execution:**

**1. compilation**

**nvcc oddevensort.c**

**2. run**

**a.out**

**Input:** Undirected weighted Graph in the form of associated weight matrix

**Output:** Minimum Spanning Tree

**Algorithm:**

Prim's algorithm Serial

1: PRIM(V, E, w, r )

2: VT := frg

3: d[r ] := 0

4: for all v 2 (V 􀀀 VT ) do

5: d[v] := w(r ; v) if (r ; v) 2 E else d[v] := 1

6: end for

7: while VT 6= V do

8: Find vertex u 2 (V 􀀀 VT ) such that

d[u] = minfd[v] j v 2 (V 􀀀 VT )g

9: VT := VT [ fug

10: for all v 2 (V 􀀀 VT ) do

11: d[v] := minfd[v];w(u; v)g

12: end for

13: end while

Parallelizing Prim's algorithm

1. d[v] is updated for all v
   1. Cannot choose two vertices in parallel
   2. Cannot parallelize outer while loop
   3. Instead we parallelize the inner for loop
2. Every process holds a block column of adjacency matrix A:
   1. A = [ A1 A2 \_ \_ \_ Ap]

and corresponding part of vector d

1. Process Pi holds vertex subset Vi
2. Owner computes: Process Pi responsible for updating its part of d
3. Find global minimum (line 8) with all-reduce
4. Update d in parallel (line 10of d

**Conclusion:**

Hence, we have successfully studied concurrent algorithm using OPENMP

**Title:**

Implement Reader-Writer problem using OPENMP

**Objectives:**

To learn multi-core and concurrent programming using OpenMP for reader writer application.

**Aim:**

To learn and implement Reader-Writer problem using OPENMP technology.

**Prerequisites (HW/SW Requirement):**

Fedora 20 (Or Any equivalent OS), GCC, OPENMP Compiler Environment

**Theory:**

**About OpenMP**

* OpenMP (Open Multi-Processing)
* an API that supports multi-platform shared memory multiprocessing *programming in C, C++, and Fortran*, on most processor architectures and operating systems
* Including Solaris, AIX, HP-UX, Linux, Mac OS X, and Windows platforms.
* It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.
* OpenMP uses a portable, scalable model
* It gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer**.**

**Introduction**

* OpenMP is an implementation of multithreading
* A method of parallelizing whereby a master *thread* (a series of instructions executed consecutively) *forks* a specified number of slave *threads* and the system divides a task among them.
* The threads then run concurrently, with the runtime environment *allocating threads to different processors.*

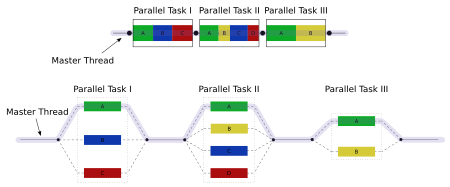


Figure: 10.1 an illustration of multithreading where the master thread forks off a number of threads which execute blocks of code in parallel.

* The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed.
* Each thread has an *id* attached to it which can be obtained using a function (called omp\_get\_thread\_num())
* The thread id is an integer, and the master thread has an id of *0*
* After the execution of the parallelized code, the threads *join* back into the master thread, which continues onward to the end of the program.
* By default, each thread executes the parallelized section of code independently.
* *Work-sharing constructs* can be used to divide a task among the threads so that each thread executes its allocated part of the code.
* Both task parallelism and data parallelism can be achieved using OpenMP in this way.
* The runtime environment allocates threads to processors depending on usage, machine load and other factors.
* The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions
* The OpenMP functions are included in a header file labelled omp.h in C/C++.

**The core elements**

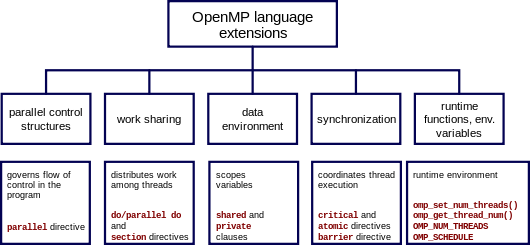
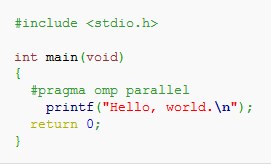


Figure 10.2 Chart of OpenMP constructs

* The core elements of OpenMP are the constructs for
  + Thread creation,
  + Workload distribution (work sharing),
  + Data-environment management,
  + Thread synchronization,
  + User-level runtime routines and
  + Environment variables.

**"Hello, world." using multiple threads**

* The **pragma *omp parallel*** is used **to fork additional threads** to carry out **the work enclosed in the construct in parallel.**
* The **original thread** will be denoted **as *master thread* with thread ID 0**
* Example (C program): Display "Hello, world." using multiple threads.



* Use flag -fopenmp to compile using GCC:



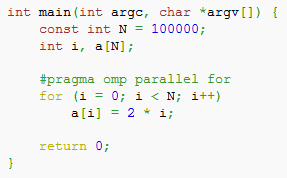
* Output on a computer with two cores, and thus two threads:



**OpenMP Constructs**

**Work-sharing constructs:**

* Used to specify how to assign independent work to one or all of the threads.
* ***omp for* or *omp do*:** used to split up loop iterations among the threads, also called loop constructs.
* ***sections*:** assigning consecutive but independent code blocks to different threads
* ***single*:** specifying a code block that is executed by only one thread, a barrier is implied in the end
* ***master*:** similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.
* Example: initialize the value of a large array in parallel, using each thread to do part of the work



**OpenMP clauses:  
Data sharing attribute clauses**

* ***shared*:** the data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.
* ***private*:** the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.
* ***default*:** allows the programmer to state that the default data scoping within a parallel region will be either *shared*, or *none* for C/C++, or *shared*, *firstprivate*, *private*, or *none* for Fortran. The *none* option forces the programmer to declare each variable in the parallel region using the data sharing attribute clauses.
* ***firstprivate*:** like *private* except initialized to original value.
* ***lastprivate*:** like *private* except original value is updated after construct.
* ***reduction*:** a safe way of joining work from all threads after construct.

**Synchronization clauses**

* ***critical*:** the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
* ***atomic*:** the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using *critical*.
* ***ordered*:** the structured block is executed in the order in which iterations would be executed in a sequential loop
* ***barrier*:** each thread waits until all of the other threads of a team have reached this point. A work-sharing construct has an implicit barrier synchronization at the end.
* ***nowait*:** specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

**Scheduling clauses**

* ***schedule(type, chunk)*:** This is useful if the work sharing construct is a do-loop or for-loop. The iteration(s) in the work sharing construct are assigned to threads according to the scheduling method defined by this clause. The three types of scheduling are:
* ***static*:** Here, all the threads are allocated iterations before they execute the loop iterations. The iterations are divided among threads equally by default. However, specifying an integer for the parameter *chunk* will allocate chunk number of contiguous iterations to a particular thread.
* ***dynamic*:** Here, some of the iterations are allocated to a smaller number of threads. Once a particular thread finishes its allocated iteration, it returns to get another one from the iterations that are left. The parameter *chunk* defines the number of contiguous iterations that are allocated to a thread at a time.
* ***guided*:** A large chunk of contiguous iterations are allocated to each thread dynamically (as above). The chunk size decreases exponentially with each successive allocation to a minimum size specified in the parameter *chunk*

**IF control**

* *if*: This will cause the threads to parallelize the task only if a condition is met. Otherwise the code block executes serially.

**Initialization**

* ***firstprivate*:** the data is private to each thread, but initialized using the value of the variable using the same name from the master thread.
* ***lastprivate*:** the data is private to each thread. The value of this private data will be copied to a global variable using the same name outside the parallel region if current iteration is the last iteration in the parallelized loop. A variable can be both *firstprivate* and *lastprivate*.
* ***threadprivate*:** The data is a global data, but it is private in each parallel region during the runtime. The difference between *threadprivate* and *private* is the global scope associated with threadprivate and the preserved value across parallel regions.

**Data copying**

* ***copyin*:** similar to *firstprivate* for *private* variables, *threadprivate* variables are not initialized, unless using *copyin* to pass the value from the corresponding global variables. No *copyout* is needed because the value of a threadprivate variable is maintained throughout the execution of the whole program.
* ***copyprivate*:** used with *single* to support the copying of data values from private objects on one thread (the *single* thread) to the corresponding objects on other threads in the team.

**Reduction**

* ***reduction(operator | intrinsic : list)*:**
* the variable has a local copy in each thread, but the values of the local copies will be summarized (reduced) into a global shared variable.
* This is very useful if a particular operation (specified in *operator* for this particular clause) on a data type that runs iteratively so that its value at a particular iteration depends on its value at a prior iteration.
* Basically, the steps that lead up to the operational increment are parallelized, but the threads gather up and wait before updating the data type, then increments the data type in order so as to avoid racing condition.
* This would be required in parallelizing numerical integration of functions and differential equations, as a common example.

**Others**

* ***flush*:** The value of this variable is restored from the register to the memory for using this value outside of a parallel part
* ***master*:** Executed only by the master thread (the thread which forked off all the others during the execution of the OpenMP directive). No implicit barrier; other team members (threads) not required to reach.

**User-level runtime routines**

* Used to modify/check the number of threads, detect if the execution context is in a parallel region, how many processors in current system, set/unset locks, timing functions, etc.

**Environment variables**

* A method to alter the execution features of OpenMP applications. Used to control loop iterations scheduling, default number of threads, etc.
* For example
* ***OMP\_NUM\_THREADS* is used to specify number of threads** for an application.

**Hello World with directives**

* A basic program that **exercises the *parallel*, *private* and *barrier* directives, and the functions omp\_get\_thread\_num and omp\_get\_num\_threads (not to be confused).**

**Reader-Writer problem:**

* The Readers and Writers problem is useful for modeling processes which are competing for a limited shared resource.
* A practical example of a Readers and Writers problem is an airline reservation system consisting of a huge data base with many processes that read and write the data.
* Reading information from the data base will not cause a problem since no data is changed.
* The problem lies in writing information to the data base.
* If no constraints are put on access to the data base, data may change at any moment.
* By the time a reading process displays the result of a request for information to the user, the actual data in the data base may have changed.
* What if, for instance, a process reads the number of available seats on a flight, finds a value of one, and reports it to the customer.
* Before the customer has a chance to make their reservation, another process makes a reservation for another customer, changing the number of available seats to zero.
* In the following sections, we will look at solutions to this problem using semaphores, monitors, and message passing.
* In the next three sections, we will discuss solutions to this problem using each of the three methods mentioned in the Solution Types section:

1. **Semaphores** can be used to restrict access to the database under certain conditions.

In this example, semaphores are used to prevent any writing processes from changing information in the database while other processes are reading from the database.

1. **Monitors** can be used to restrict access to the database.

In this example, the read and write functions used by processes which access the database are in a monitor called *ReadersWriters*.

If a process wants to write to the database, it must call the *writeDatabase* function.

If a process wants to read from the database, it must call the *readDatabase* function.

Remember that monitors use the primitives **Wait** and **Signal** to put processes to sleep and to wake them up again.

In *writeDatabase*, the calling process will be put to sleep if the number of reading processes, stored in the variable *count*, is not zero.

Upon exiting the *readDatabase* function, reading processes check to see if they should wake up a sleeping writing process.

1. **Message Passing** is used to prevent a writing process from writing into the database while another process is either reading or writing.

The following solution assumes that readers and writers send their requests to a database server process that ensures that the above property holds.

**Steps for Execution:**

## Enabling OpenMP

* To activate the OpenMP extensions for C/C++ and Fortran, the compile-time flag -fopenmp must be specified.
* This enables the OpenMP directive #pragma omp in C/C++ and !$omp directives in free form, c$omp, \*$omp and !$omp directives in fixed form, !$ conditional compilation sentinels in free form and c$, \*$ and !$ sentinels in fixed form, for Fortran.
* The flag also arranges for automatic linking of the OpenMP runtime library (Runtime Library Routines).
* A complete description of all OpenMP directives accepted may be found in the OpenMP Application Program Interface manual, version 4.0.

1. Open Terminal
2. Switch to root super user
3. Then type following command:

$ gcc readwrite.c –fopenmp

**Conclusion:**

Hence, we have successfully studied multi-core and concurrent programming using OpenMP for reader writer application..