International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 2 Issue 4, April - 2013

Analyzing The Overall Performance Of N-Computing Device At Different Load Conditions

Snigdha Srivastava[1] Saumya Srivastava[2] Sneha Mani Tripathi[3]

[1, 2 & 3] *Department of computer science and Engineering, Institute of Technology and Management, GIDA, Gorakhpur, India*

Abstract

The personal computer has a very high processing power now-a-days than actually required for a single user system. Hence, it can be effectively used as a multi-user system which serves several users concurrently. This can be achieved by using N-computing device. Any N-computing device uses the processing power of a single CPU to serve multiple users. This concept is related to the concept of mapping between user level thread and kernel level thread. Therefore the performance of the device should be such that it utilizes maximum power of the system. This paper provides the method and the outcome of the performance analysis of the device under various load conditions. This performance factor has been measured in various parameters. We have also shown 100 percent utilization of CPU, i.e. utilization of full processing power. This is done with the help of some codes. In other words this paper studies how to utilize the full processing power of a personal computer for number of users simultaneously and hence save energy.

Keywords**:** *Kernel, mapping, n-computing device, performance factor, thread.*

1. Introduction

We've all become accustomed to the PC model, which allows every user to have their own CPU, hard disk, and memory to run their applications, but personal computers have now become so powerful that most people can't possibly use all the processing power they purchase. N-Computing desktop virtualization is a modern take on the time-honored concept where multiple users share the processing power of a single computer. This approach has several advantages over the traditional PC model, including lower overall costs, better energy efficiency, and simplified administration. N-computing is technology that allows multiple users to share single computer simultaneously; this means that with n-computing we could have one ordinary desktop computer catering for people or more at the same time. N-computing is a desktop virtualization company that manufactures hardware and software to create virtual desktops (sometimes called zero clients or thin clients) which enable multiple users to simultaneously share a single operating system instance. The effectiveness of parallel computing depends to a great extent on the performance of the primitives that are used to express and control the parallelism within programs. It exhibit poor performance if the cost of creating and managing parallelism is high. Even a fine-grained program can achieve good performance if the cost of creating and managing parallelism is low. Kernel level threads are, effectively, processes that share code and data space, where user level threads are implemented at the application level. This research divides responsibility for thread management between the kernel and application. Multithreading has emerged as a leading paradigm for the development of applications with demanding performance requirement. When number of users increase, so does the number of processes and hence number of threads, then performance is a major issue. Therefore analyzing the performance and trying to maximize it is the best option which is discussed along with the methods further in this paper.

2. Thread Management

Threads are the vehicle for concurrency in many approaches to parallel programming. Threads can be supported either by the operating system kernel or by user-level library code in the application address space, but neither approach has been fully satisfactory [5]. A thread is a light-weight process. The implementation of threads and processes differs

from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. There are two commonly used thread models: kernel level threads and user level threads. Kernel level threads suffer from the cost of frequent user-kernel domain crossings kernel scheduling priorities. User level threads are not integrated with the kernel; blocking all threads whenever one thread is blocked. On a single processor, multithreading generally occurs by time-division multiplexing the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor (including multi-core system), the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task. The kernel of an operating system allows programmers to manipulate threads via the system call interface. Some implementations are called a kernel thread, whereas a lightweight process is a specific type of kernel thread that shares the same state and information. Multithreading has emerged as a leading paradigm for development of application with demanding performance requirements [1]. Therefore we can say that generally, threads are located on shared data structure: a shared run queue for ready threads and shared communication structures for blocked thread. Access control to the shared resource is maintained through lock-based mechanism which ensures safe access to such critical section of core [2].

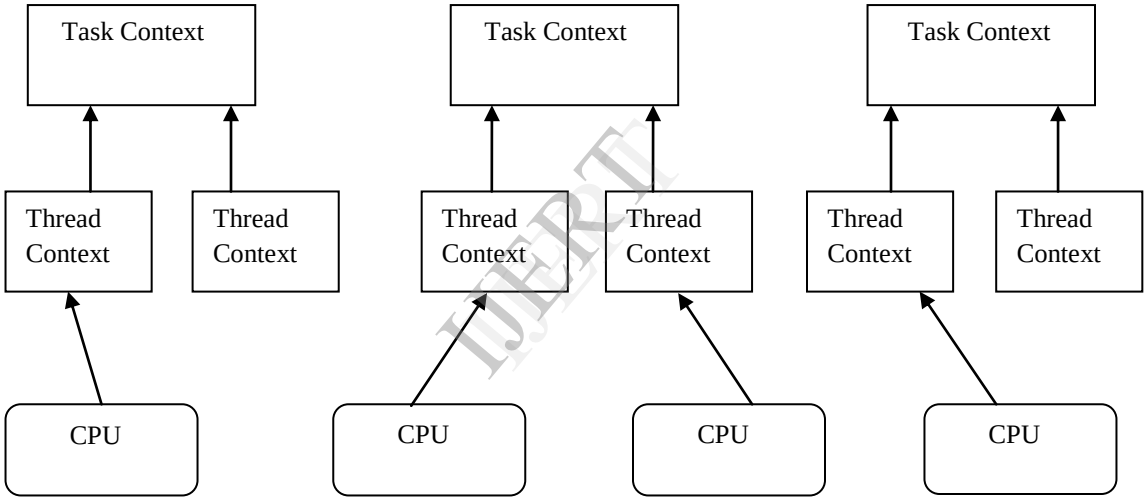

Figure 1. Thread Management

Kernel level threads share some of the disadvantages of processes. Switching between them is slow, taking an order of magnitude more time than a user level thread context switch. Also they are scheduled by the kernel; with no application control .this can be negatively affect performance. For example, if threads have different priorities and the priorities are not visible to the kernel, a low priority thread may be scheduled in place of a high priority one. User level threads systems control scheduling decisions, but because they are not integrated with the kernel, when one thread blocks (e.g. to perform I\O), all of the user level threads sharing the process are blocked. This advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores or across a cluster of machines—because the threads of the program naturally lend themselves to truly concurrent execution. In such a case, the programmer needs to be careful to avoid race conditions, and other non-intuitive behaviors. The Scheduler Activations model, proposed by Anderson et al., combines CPU allocation decisions with application control over thread scheduling. It discusses the performance characteristics of an implementation of Scheduler Activations for a uniprocessor system, and proposes an analytic model

for determining the class of applications that benefit from its use [3].

In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require mutually exclusive operations (often implemented using semaphores) in order to prevent common data from being simultaneously modified, or read while in the process of being modified. The operating system kernel has complete control over the allocation of processors among address spaces including the ability to change the number of processors assigned to an application during its execution. To achieve this, the kernel notifies the address space thread scheduler of every kernel event affecting the address space, allowing the application to have complete knowledge of its scheduling state. . For fixed priority scheduling on uniprocessors, under certain conditions, the system's schedulability is maximized when the priorities are chosen in the inverse order of the task's deadlines [4]. The thread system in each address space notifies the kernel of the subset of user-level thread operations that can affect processor allocation decisions, preserving good performance for the majority of operations that do not need to be reflected to the kernel [5]. Today's operating systems provide kernel threads for parallel applications and multi-threaded servers. Scheduling plays an important role with regard to efficiency and fairness especially for distributed applications, multimedia processing and server processes. A multi-threaded application should be able to specify the scheduling strategy for its threads itself. In most modern operating systems the scheduling strategy is hard-coded into the kernel and cannot be changed by the user. There are a few user-level thread packages available, where the users can define the scheduling strategy. Yet user-level threads are not suitable for applications that interact with the operating system frequently, such as server processes or distributed applications. Each application can have one or more of its own schedulers, which can define the application-specific scheduling strategy. Thus, the programmer can implement his own scheduling strategy for his application or even for subsystems inside the application [6].

3. Scheduling and mapping of threads by kernel

In most computer operating systems, the kernel is the central component. It is the bridge between the user and applications and the computer hardware. It also is the mechanism that allows the computer to handle multiple users and multiple tasks simultaneously. The types of kernels are the monolithic kernel, the microkernel, the hybrid kernel, the nanokernel and the exokernel. The kernel manages all of the computer's system resources. This includes long-term storage, the central processing unit (CPU), short-term memory and the input and output devices. When an application needs one of these resources, the kernel makes the resource available and completes the request. This handling of resources allows the operating systems to be both multi-user and multitasking. The operating system does not actually perform more than one task at a time. Instead, the kernel switches tasks at such a high speed that the computer appears to be performing multiple tasks. The kernel also is responsible for making sure that resources used by one user or process are not violated the request of another user or process. There two main types of kernels are the monolithic kernel and the microkernel. Monolithic kernels employ a supervisory method of resource management in which all of the operating system services are run in the same address space, called the kernel space. Some monolithic kernels can load and unload executable modules. This extends the operating system's capabilities while still maintaining a minimum amount of code running in the kernel space at any one time. Micro kernels run only the minimal amount of operating system services, such as memory management, thread management and inter-process communication in the kernel space. All other services, such as device drivers, user interfaces and file management, are run in the user space. The microkernel severely minimizes the amount of code that is running in the kernel mode.

Every thread has a thread priority assigned to it. Threads created within the common language runtime are initially assigned the priority of ThreadPriority.Normal. Threads created outside the runtime retain the priority they had before they entered the managed environment. Threads are scheduled for execution based on their priority. Even though threads are executing within the runtime, all threads are assigned processor time slices by the operating system. The details of the scheduling algorithm used to determine the order in which threads are executed varies with each operating system. Under some operating systems, the thread with the highest priority (of those threads that can be executed) is always scheduled to run first. If multiple threads with the same priority are all available, the scheduler cycles through the threads at that priority,

giving each thread a fixed time slice in which to available to run, lower priority threads do not get to execute [5]. Therefore the sequence of execution of threads is totally dependent on its priority. When there are no more runnable threads at a given priority, the scheduler moves to the next lower priority and execute. As long as a thread with a higher priority is schedules the threads at that priority for execution. If a higher priority thread becomes runnable, the lower priority thread is preempted and the higher priority thread is allowed to execute once again.

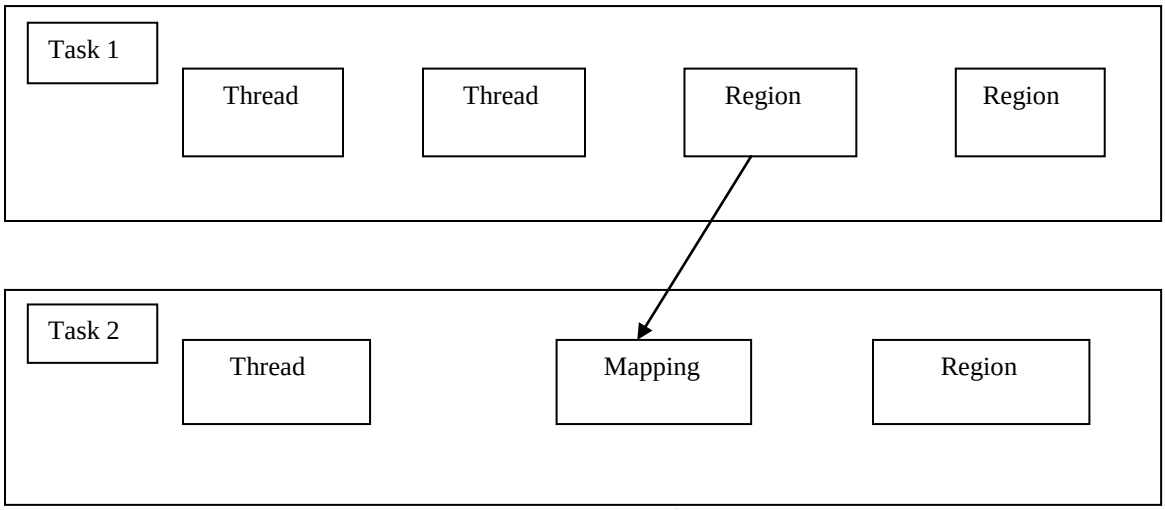

Figure 2. Thread Mapping between Kernel level Thread and User level Thread

The operating system kernel provides each user-level thread system with its own virtual multiprocessor, the abstraction of a dedicated physical machine except that the kernel may change the number of processors in that machine during the execution of the program [5].

There are two types of threads to be managed in a modern system: User threads and kernel threads. User threads are supported above the kernel, without kernel support. These are the threads that application programmers would put into their programs. Kernel threads are supported within the kernel of the OS itself. All modern OS support kernel level threads, allowing the kernel to perform multiple simultaneous tasks and/or to service multiple kernel system calls simultaneously. In a specific implementation, the user threads must be mapped to kernel threads, using one of the following strategies:

- **Many-To-One Model:**

In the many-to-one model, many user-level threads are all mapped onto a single kernel thread. Thread management is handled by the thread library in user space, which is very efficient.

- **One-To-One Model:**

The one-to-one model creates a separate kernel thread to handle each user thread. One-to-one model overcomes the problems listed above involving blocking system calls and the splitting of processes across multiple CPUs.

- **Many-to-Many Model:**

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.

4. Methods and code for analysis

The maximum CPU utilization can be achieved by increasing the number of processes per CPU cycle. This task can be achieved by running various programs, capable of creating thousands of threads. We can also use a linked list program which can create thousands of new nodes. These programs will help in achieve our goal that is to utilize full processing power. Before going to the code we should also know, how the N-computing device works that if a program is run on 1 of the nodes connected to the device then it affects the performance of other nodes too. The unique N-Computing technology is composed of three primary components: vSpace virtualization software, a user extension protocol, and access devices. By combining all three of these components into an integrated solution, N-Computing delivers unmatched performance at an incredibly low cost. Traditional

thin client solutions and other PC alternatives all rely on separate components from disparate vendors, resulting in sub-optimal performance at higher costs.

- **VSpace Server software:**

N-Computing vSpace Server virtualization software, included with the hardware, creates the virtual desktops in the shared PC by dividing the computer's resources into independent sessions that give each user their own PC experience. The vSpace software uses the company's proprietary display and communications protocol ("UXP") to communicate between the shared computer and the user stations.

- **Desktop virtualization software:**

The virtualization software divides the computer's resources into independent sessions that give each user their own rich PC experience. Functioning as a data manager, it transmits and handles the desktop display and remote activities from the users' keyboard and mouse (through the access device).

- **Extension protocol:**

A key part of being able to deliver a full remote computing experience is the extension protocol used. Traditional thin clients use protocols that were developed for occasional use by administrators for temporary remote control. N-Computing developed its unique User extension Protocol (UXP) for continuous use by end users demanding a full PC experience. As a result, multimedia applications including streaming video, Flash, and 3D graphics can be supported.

- **Access devices:**

The N-Computing access devices do not use PC-based processors or chipsets and do not run a local operating system. All of the primary functionality is integrated into a single chip that has an optimal set of resources for working with the N-Computing virtualization software and extension protocol. This System-on-Chip (SoC) contains patented technologies for delivering unmatched performance from a very low-power device.

Some codes were implemented on 5 nodes connected via N-computing device, in order to utilize its full processing power. These programs are written in c and java programming languages. These codes are implemented on one node yet they affected the other nodes too. On running these codes we analyzed the performance of the device, simply by varying the load applied on the device. **Some of the implemented codes are as following:**

4.1 Code to generate thousands of threads

```
class demo1 extends Thread
{
public void run()
{
for(int i=1;i<=156000;i++)
if(i%2==0)
System.out.println(i +"is Even");
else
System.out.println(i +"is odd");
}
}
class demo2 extends Thread
{
int limit=156892;
int num=1;
public void run()
{
for(int i=1;i<=limit;i++)
{
int sum=0;
while(i>0)
{
sum=sum+(i%10);
i=i/10;
}
}
}
}
 class abc
{
public static void main(String ar[])
{
int i;
for(i=0;i<20000;i++)
{
new demo1().start();
new demo2().start();
}
}
}
```

4.2 Code to generate multiple nodes in a linked list

```
#include<alloc.h>
struct node
{
int item;
struct node *next;
}*START=NULL;
void add(int x)
{
struct node *temp1,*temp2;
if(START==NULL)
{
temp1=(struct node *) malloc(sizeof(struct node *));
temp1->next=NULL;
temp1->item=x;
START=temp1;
}
else
{
```

```
temp1=START;
while(temp1->next!=NULL)
temp1=temp1->next;
temp2=(struct node *) malloc(sizeof(struct node *));
temp2->next=NULL;
temp2->item=x;
temp1->next=temp2;
}
}
void display()
{
struct node *temp;
if(START==NULL)
printf("\n No item to display");
temp=START;
while(temp!=NULL)
{
printf("%2d",temp->item);
temp=temp->next;
}
}
void main()
{
clrscr();
while(!kbhit())
add(10);
display();
getch();
}
```

5. Performance measure in N-computing environment

The overall performance of the device is measured through various parameters. These parameters collectively judge the performance and are helpful in comparing the results of the device on varying the load. Such as we can measure the difference in time units taken by the device or the number of registers used or cache utilization. These factors are as follows:

- **CPU Usage:**

CPU usage represents how much of the central processing unit's total ability to manipulate data is being used. There are separate graphs for each processing core that our computer can use. When our computer is idling, and we have no active programs running, the CPU usage is typically at 1 percent.

- **CPU time:**

CPU time is the amount of time for which a central processing unit was used for processing instructions of a computer program, as opposed to, for example, waiting for input/output (I/O) operations. The CPU time is often measured in clock ticks or seconds. CPU time is also mentioned as percentage of the CPU's capacity at any given time on multi-tasking environment. This is also referred as CPU usage or process time.

- **Paging File Usage:**

Our paging file is a software support for our computer's normal memory-using hard drive space. On older operating systems, this was commonly referred to as virtual memory.

- **Commit charge:**

This measures the amount of 'committed virtual memory' in the system. This is all memory requested by processes that is not backed by some named file. One way to look at this is that the system has a certain budget for virtual memory, and each program request is charged against that budget. The Total commit charge is the current in-use value; the Limit is the sum of the page file sizes and the physical memory that's available in principle for programs .

- **Physical memory:**

Physical memory is representative of the actual physical memory located within our computer. The physical memory graph tracks how much available memory vs. used memory our computer has available. When physical memory is low, it will start transferring data to the paging file to supplement itself.

- **Kernel memory:**

The kernel is the operating device between applications and the hardware units of a computer such as the CPU. This is one of the most basic components of computer hardware and controls the interaction of all other devices. Kernel memory is not only physical memory, but paged memory as well. When the kernel memory becomes full, it often causes fatal crashes in programs or the entire computer.

- **Number of handles, processes and threads:**

The kernel supplies programs with 'objects' such as files, shared-memory sections, registry keys, and so on. A program uniformly manipulates an object by means of a handle, which is a temporary connection to the object. A handle is not the object; for example, if a file is opened for 17 different uses at the same time, it will have 17 different handles connected to it. A process is an instance of a program in execution. If we're running Explorer 3 times, then there will be 3 processes running. The program is the thing that persists - the program we had yesterday is the program we have today (unless we did something!). Processes come and go. Each process is made up of one or more threads, at the decision of the programmer.

5.1 Snapshots comparing the performance

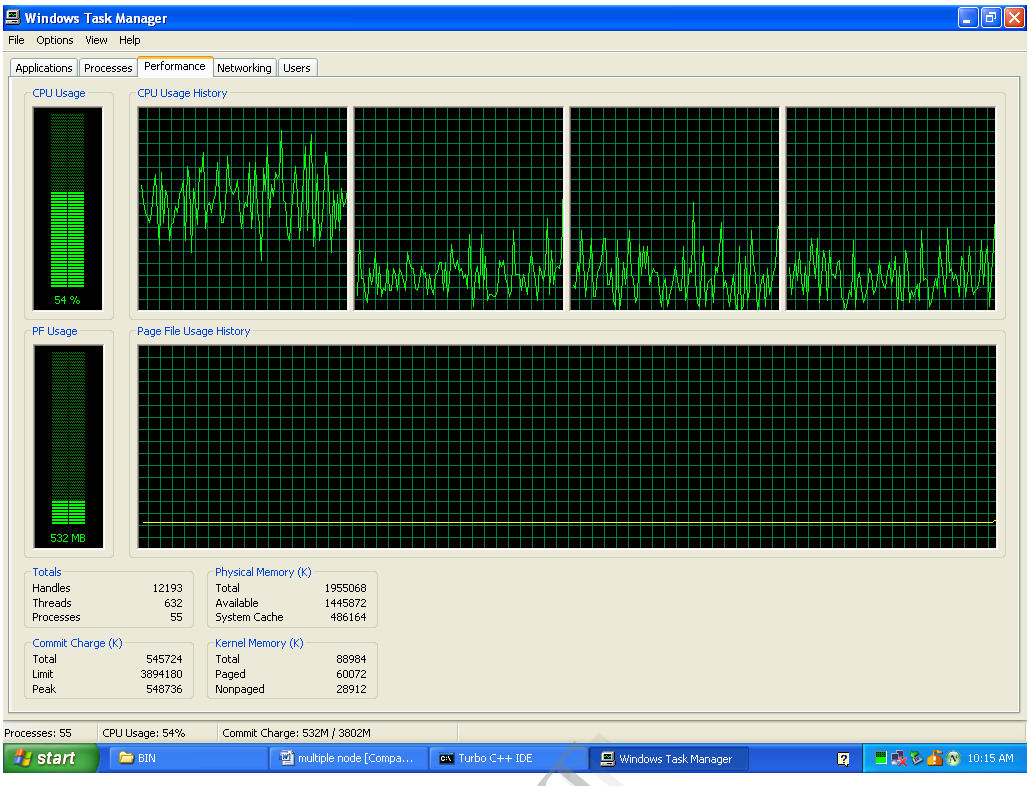

Figure 3. Performance on creating multiple nodes at high load

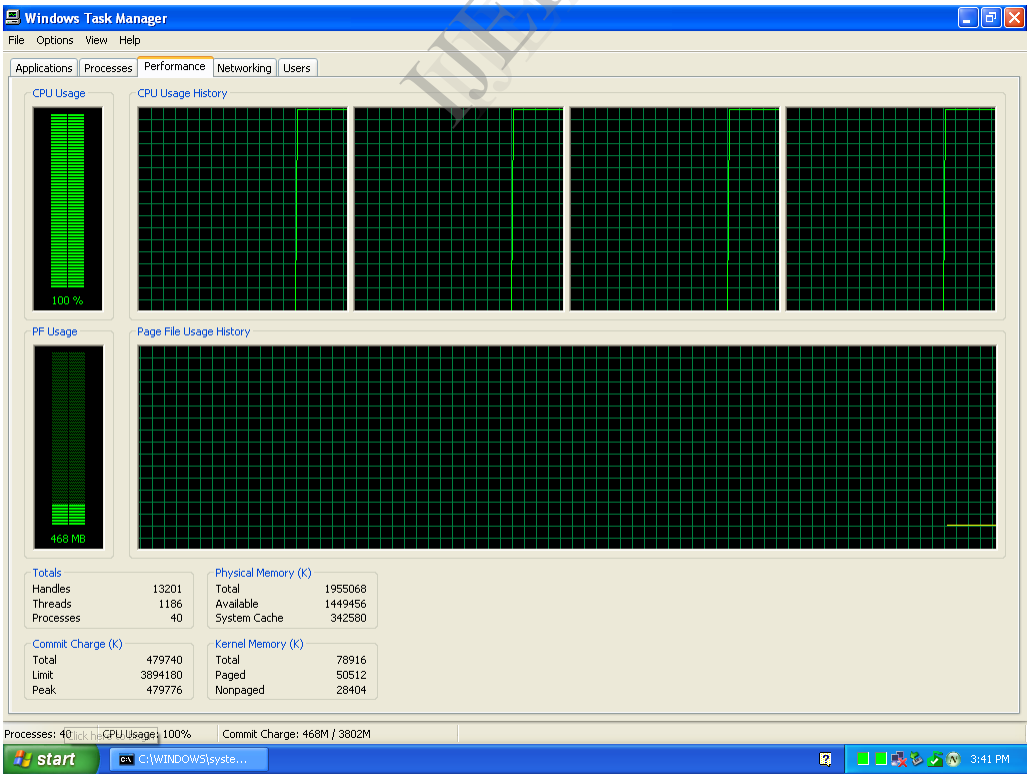

Figure 4. Performance on creating several threads at a different load

The above shown figures show the performance outcome on running different-different programs at different load i.e. at the times when different-different number of nodes is connected to the device. As we can see in the snapshots that number of processes, registers used, Kernel memory vary. For example System cache increases at high load and so does the peak commit charge, whereas some factor increase on the basis of time such as PF usage. The CPU utilization reaches its maximum abruptly when threads are created otherwise it takes time and vary in its utilization capacity.

6. Conclusion

N-Computing is a modern take on the time-honored concept where multiple users share the processing power of a single computer. This approach has several advantages over the traditional PC model like: allowing organizations to use a single desktop PC like a minicomputer, offering simplicity, delivering power at a highly affordable price, ensuring long term customer and partner success. Our paper analyzes device's performance and checks if full processing utilization can be achieved or not at the n-computing environment. It also studies the effect of increase and decrease of load conditions on the device. This effect can prove useful in terms of increment in performance for organizational purpose and in this way it could be an effective medium of full power utilization and energy savings, that also at reasonable costs.

7. References

1. K. Schwan and H. Zhou [Georgia Institute of Technology], *"Dynamic scheduling of hard real-time tasks and real-time threads",* IEEE transactions on Software Engineering, Aug. 1992, *Volume 18*, *Issue 8*, p. 736-748.
2. K.Debattista, K.Vella and J.Cordina, *"Wait-free cache-affinity thread scheduling",* IEEE Proc., Softw.- April 2003, *Volume 150*, *Issue 2*, p. 137-146.
3. Christopher Small and Margo Seltzer, *"Scheduler Activations on BSD:Sharing Thread Management Between Kernel and Application"*, Harvard University.
4. L. Mangeruca, A. Ferrari and A. L. Sangiovanni-Vincentelli, *"Uniprocessor scheduling under precedence constraints", RTAS'06, p. 157-166*, IEEE Computer Society, Washington DC, USA.
5. Thomas E. Anderson, Brian N. Bershad and others**,** *"Schedular activations: effective kernel support for the user-level management of parallelism",* ACM Transactions on Computer Systems [New York, USA], Feb. 1992, *Volume 10*, *Issue 1*, p. 53-79.
6. Thomas Riechman, Jurgen Kleinoder, *"User-Level Scheduling with Kernel Threads",* Department of Computer Science, University of Erlangen-Nurnberg, Germany.
7. D. Kang and J.L.Gaudiot [University of California], *"Speculation- aware thread scheduling for simultaneous multithreading"*, Electron. Lett. 4 March 2004, *Volume 40, Issue 5*, p. 296-298.