Chapter-7

SEMI-DECENTRALIZED TASK SCHEDULING STRATEGY FOR DISTRIBUTED IMAGE COMPUTING SYSTEM

INTRODUCTION

Ray tracing is one of many techniques for rendering three-dimensional images. Ray tracing is a rendering model, which is based on the physics of light and how it interacts with different materials. Ray tracing gets its name from the use of simulated rays of light in producing images. In ray tracing, visible surface determination, reflection, refraction, and shading are done using physically based approximations of the way real light behaves. Ray tracing is capable of rendering complex mathematical surfaces, multi-dimensional vector fields, and discrete polygonal meshes. Classical sequential ray tracing algorithms can be adapted for parallel execution environments, and are well suited for the production of photorealistic images. Other rendering techniques that are computationally cheaper are typically limited to the use of polygonal meshes for modeling objects, and can be much more difficult to parallelize efficiently. Ray tracing is well suited to parallel computation. By parallelizing the rendering process, execution time can be reduced by more than two orders of magnitude, given appropriate computational resources. However, the algorithm is quite computing intensive and the demand on processing power grows as the scenes to visualize become more and more complex. The applications, such as scientific visualization, CAD, vehicle of simulation, and virtual reality can require hundreds of MFLOPS computing power that is in many cases, far beyond the capabilities of a single WS/PC. It has been realized that massively parallel computers can be an effective way to accelerate these computations.

Due to the new advancements in distributed computing environment it is possible to process such kind of images in reasonable time on modest budgets. For better utilization of the computational powers of all WSs/PCs in a HDC system, the efficient application partitioning and scheduling is needed. Ray tracing is a powerful rendering technique that can produce high-quality graphics images; however, this quality comes at a price of intensive calculation and long rendering times. Even relatively simple ray-traced animations can prohibitively expensive to render on a single processor. For longer, more complex animations, the rendering time can be intractable. Fortunately, ray tracing is a prime candidate for parallelization since its processing is readily amenable to subdivision. Specifically, ray tracing inherently contains a large amount of parallelism due to the independent nature of its pixel calculations Whit [10]; therefore, most ray tracing rendering algorithms lend themselves to parallelization in screen space.

In heterogeneous distributed computing system, both the static and dynamic (RTS) strategies can used to balance the loads. Since WSs/PCs have performance variation characteristic [1, 2], the static task distribution is not effective for HDC system [3, 4]. RTS strategy can achieve nearly perfect load
balancing [5]. The performance variations and non-homogeneous nature of the application (image) are adjusted at runtime [6]. The RTS strategy performance depends upon the size of the sub-task. If sub-task size is too small then it generates a serious inter-process communication overhead. In other case, if the sub-task size is too large then it may create a longer master/client waiting time due to the inappropriate sub-task size of slow WS [6]. Many researchers suggest enhancement in adaptive task scheduling strategies for homogeneous distributed systems [7-9]. However, these strategies do not work well for HDC systems. The crucial point is that these are based on fixed parameters that are tuned for the specified hardware. In heterogeneous systems, this tuning is often not possible because both the computational power and the network bandwidth are not known in advance, which may change unpredictably during runtime.

The strategies are based on task distribution and then task migration from heavily loaded to light-loaded workers is expressed in [1]. The task migration has two serious drawbacks [10-13]:

- All workers should continuously monitor the status of other workers.
- During the computations, a worker has to monitor its load and float the information on the network. This produces a large amount of communication overhead.

In contrast to above, the adaptive strategy is developed for heterogeneous distributed image computing system. The strategy should not be complex; their overhead involved should not negate the benefit, and should support large HDC systems. The adaptive strategy is based on “Worker (server) Initiated Sub-task sizing (WIS) strategy”.

In WIS strategy, each worker has authority to estimate its next sub-task size without needing other workers state information and suggests its sub-task size to the master WS/PC. Since the HDC system environment is commonly available and it is rapidly growing the performance verses scalability is important. Therefore, in this chapter, the semi-decentralized resource management and decision about the task scheduling on HDC system are practically examined.

### 7.1. TASK PARTITIONING AND SCHEDULING STRATEGIES

The goal of the program partitioning and scheduling is to minimize the overhead caused by inter-process communication time, while preserving the highest possible degree of parallelism.

**Runtime Task Scheduling (RTS) strategy**

A unit of task (one horizontal scan-line of the image) is distributed at runtime. As the worker completes the previous assigned sub-task, new sub-task is assigned for processing. Viewing Fig. 1 can easily differentiate the task mapping mechanism of RTS.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>Total number of tasks (640 horizontal scan-lines of the image).</td>
</tr>
<tr>
<td>$i$</td>
<td>Worker number.</td>
</tr>
<tr>
<td>$t_{(i)}$</td>
<td>Worker’s assigned task or computed sub-task size.</td>
</tr>
<tr>
<td>$f_{initial(i)}$</td>
<td>Worker’s initial sub-task size.</td>
</tr>
<tr>
<td>$T_b$</td>
<td>Total number of unprocessed tasks. $(T_b = T - t_i)$</td>
</tr>
<tr>
<td>$W_{p(i)}$</td>
<td>Worker’s estimated current performance.</td>
</tr>
<tr>
<td>$W_{pu(i)}$</td>
<td>Worker’s performance at unloaded condition.</td>
</tr>
<tr>
<td>$W_{pmin}$</td>
<td>Slowest worker’s performance at unloaded condition.</td>
</tr>
<tr>
<td>$W_{q(i)}$</td>
<td>Worker’s job queue length at unloaded condition.</td>
</tr>
<tr>
<td>$W_{q(i)}$</td>
<td>Worker’s current job queue length.</td>
</tr>
</tbody>
</table>

Table 1: Symbols used and their definitions

Worker Initiated Sub-tasking (WIS) Scheduling Strategy

For centralized configuration, state information gathering and making decision about the scheduling of task is seen to be saturated when HDC system consists of hundreds of WSSs/PCs. The master has to collect each worker’s state information and decide the size of the task for next assignment. Therefore, a novel approach of semi-decentralized resources collections is proposed. In that is, each worker has autonomy to decide next sub-task size according to its current performance. The WIS strategy is designed by considering the following points in mind:

- The adaptive nature of task sizing scheduling strategy is essential to absorb the heterogeneity and performance variation characteristics of the worker.
- The worker should compute its sub-task size according to its current performance without knowledge of other workers’ performance.

This approach cuts the waiting time, which would occur otherwise due to collection of other workers’ state information [14]. Effectively, this reduces the network traffic, as every worker floats its performance information on the network in a specified interval. For making decision about the task partitioning and scheduling the researchers use benchmark programs to monitor the performances
in a HDC system. Due to the non-homogeneous nature of application (image), the PC's/WS's performance estimated from worker's runtime responses may not be the accurate [15]. The authors in [16] clearly indicate that, the CPU job queue length approach for load balancing gives better results. In WIS strategy, the CPU jobs queue length information for estimating the worker’s performance is used. The estimated worker’s performance is used for worker’s sub-task computation.

![Diagram](image)

**Fig. 1:** Tasks mapping among four workers: a) RTS and b) adaptive WIS strategies.

In this strategy, the manager (master) has pre-knowledge of each worker’s performance and its job queue length information at unloaded condition (no other user was logged in the network). Manager utilizes this information to compute the initial sub-task size for each worker in HDC system using equation 1. The slowest worker’s sub-task size is supposed to be unit of task (e.g., one horizontal scan-line of the image).

\[
t_{\text{initial}(i)} = \frac{W_{\text{pu}(i)}}{W_{\text{puslow}}}
\]

Each worker of the HDC system records his initial sub-task size \( t_{\text{initial}(i)} \). The worker also has knowledge of its performance and job queue lengths information at unloaded condition. Whenever the worker finishes the task, it estimates its current performance with reference to its noted performance and queues length at unloaded condition using equation 2.

\[
W_{\text{pu}(i)} = \frac{W_{\text{pu}(i)} * W_{q(i)}}{W_{\text{quload}(i)}}
\]  

(2)

Further, worker estimates its next sub-task size using equation 3:

\[
t_{(i)} = \frac{W_{\text{pu}(i)} * t_{\text{initial}(i)}}{W_{\text{pu}(i)}}
\]  

(3)
Note that, in WIS strategy, each worker’s sub-task size is modified only when worker’s performance ratio:

\[
\frac{Current}{Previous} \approx \frac{W_{p(t)}}{W_{p(t-1)}} \quad \text{or vice versa becomes} \quad 0.25,
\]

because the rapidly fluctuating task sizing never gives better results for HDC system [17]. The 70% of the total task is scheduled according to the above described procedure and for remaining; the manager reduces the sub-task size \( t_{(a)} \) of each worker by 30% on each worker’s request. This reduces the work load-imbalance.

To achieve a better load balancing and avoid master waiting time at the end of the application, the following components are included in this strategy.

**Slow worker omission**

The heterogeneous distributed computing system is composed of cooperative computing resources. Its performance also depends upon the other users application load. In some cases, the WS’s performance degrades/not available, this creates a long waiting time for the master/client machine at the end of the parallel application when degrade the HDC system performance. To avoid such type of waiting, the following checkpoint is included in proposed strategy:

After 70% of the total task completion, the manager checks those workers, which still have not completed the first assigned task. The manager marks that worker “as not available” in the current HDC configuration and adds its assigned task \( t_{(i)} \) to the unprocessed task balance( \( T_{b} \) ). It eliminates excessive manager waiting time that may occur at the end of application.

**Sub-task duplication**

When all the tasks of the application are distributed to the workers no task is left to assign further to only remaining idle worker. The faster worker finishes the last assigned task earlier as compared to the slow workers; the slow worker very often creates a manager waiting time. To avoid such type of waiting, the following slow worker’s task duplication mechanism in this strategy includes:

The manager searches the least performance worker in the current HDC configuration and duplicates its assigned task to the next idle worker. It creates a competition between two workers, the result of the worker who completes the task first is considered and that of the other is ignored. Each least worker’s task is duplicated only once. So all computing resources are utilized at the end of application. This also eliminates the excessive manager waiting time that would have occurred at the end of application.

**7.1.1. Workers Sub-task Sizing**

Before implementing the WIS strategy on actual HDC system, the sub-task sizing behaviors of workers are simulated. For this purpose, separate C codes are written to feed the pre-measured machine’s performance at compile time. Let us,
consider the case when the HDC system is composed of number of workers (NM) = 20. The two higher performance machines sub-task sizing (HPM 20-1 & HPM 20-2) behaviors are listed in Fig. 2.

Fig. 2: The sub-task size modification in upper two higher performance workers WIS strategy

It is observed from Fig. 2, for all workers’ requests the sub-task size of first higher performance worker (HPM 20-1) is always larger than the sub-task size of second higher performance worker (HPM 20-2). In WIS strategy, each worker’s sub-task size starts from its peak value and saturates after certain number of requests, as workers sub-task size decreases on each worker’s request at the end of the application.

7.2. IMPLEMENTATION

The performances of RTS and WIS strategies are mainly evaluated in terms of speedup and workers idle time cost [23]. Here these terms are defined briefly.

Speedup (SP): The speedup is used to quantify the performance gain from a parallel computation (\( T_{pf} \)) of an application over its computation (\( T_{pdf} \)) on a single fastest machine in the network of WSs/PCs.

\[
\text{Speedup of HDC system (SP)} = \frac{T_{pf}}{T_{pdf}}
\]  

Workers idle time cost:

The activities of worker’s process in a HDC image processing system are mainly composed of three factors:

- worker setup time to load pattern data file and initialize all programming parameters,
worker raytracing computation time,

- worker time taken to report result (data) and getting new task from the master. The number of requests made by the workers directly affects the value of the third factor. This overhead in time is computed as:

\[ O_{(i)} = (\text{Worker's new task starting time}) - (\text{worker's previous task completion time}) \]  

where \( O_{(i)} \) is the overhead time, which includes all communications data access delays to start the processing of new task on an idle worker.

The workers idle time cost can then be expressed as:

\[ \text{Workers idle time cost} = \sum_{i=1}^{W_f} O_{(i)} \]  

where \( W_f \) is the total number of workers in HDC configuration.

The HDC system used in our investigation is composed of seven Sun workstations (WSs) loaded with SunOS/Solaris, and thirteen PCs (Intel based machines) loaded with Linux/FreeBSD operating system; all of these machines are connected via Ethernet. The network communication is handled by Open Consortium Remote Procedure Call (ONC-RPC) library and XDR filters. UNIX heavy weight processing technique is used to control many workers at the manager’s end. The investigation is carried out on raytracing images, which can be parallelized without complex inter-processors communication.

To analyze the performances of the proposed WIS strategy, we used four distributed raytracing images that were known as scene A, B, C and D. Each image is composed of 840 x 640 pixels. Fig. 3 gives the impressions of the images. The five HDC configurations are set, i.e., the HDC system is composed of number of machine (NM) = 20, 16, 12, 8 and 4. These HDC configurations were configured in such a way that (starting from the maximum workers, i.e., NM=20) each next HDC configuration is arranged by omitting the four slowest workers from the current configuration.

<table>
<thead>
<tr>
<th>Image scene A</th>
<th>Image scene B</th>
<th>Image scene C</th>
<th>Image scene D</th>
</tr>
</thead>
<tbody>
<tr>
<td>63 sec</td>
<td>122 sec</td>
<td>152 sec</td>
<td>244 sec</td>
</tr>
</tbody>
</table>

Table 2: The single fastest machine in the network image scenes processing time.
7.3 RESULTS AND DISCUSSION

The performances of WIS and RTS strategy are compared in terms of speedup, number of request made by the master and workers idle time costs. The investigated HDC system configurations have high heterogeneity in WS's performances; some machines have a speed of 400Mhz and other has 40Mhz. The processing time taken to process the image scenes A, B, C and D by the single fastest machine in the network are shown in Table 2. It is clear that image scene A is lightly sensed and takes less processing time, while scene D is the heavily sensed among all scenes.

Runtime Task Scheduling (RTS) strategy

The RTS strategy has a potential to absorb the machine's performance variation characteristics and non-homogeneous nature of the application at runtime tasks assignment. The number of tasks processed by the worker is proportional to its runtime performance. However, RTS strategy has two serious drawbacks as are explained in section 7.1. The performance of RTS strategy is evaluated using the fixed unit of task (one horizontal scan-line of the image). The measured speedups of this strategy are listed in Table 4. It has, at average, 29% degradation with respect to WIS strategies, respectively. Due to the large number of master's requests occurred in RTS strategy (see Table 3), more workers idle time cost is generated (see Fig. 4). As shown Fig. 4, the average measured workers idle time cost increases as the number of workers increases in HDC configurations. This is due to the following reasons:

As the number of workers increases, the waiting child processes increase at the master machine, which increases the auxiliary workload at master machine. Therefore, it may decrease its task parallelization capability, which effectively increases the worker’s waiting time.

As the number of workers increases in master and workers HDC system model, the low bandwidth network usage will increase, because every worker has a responsibility to report the results to the master. Therefore, it creates auxiliary load on the network, which may be the cause of long workers idle time cost.

(a)  (b)  (c)  (d)

Fig. 3: Raytracing images: a) scene A, b) scene B, c) scene C, and d) scene D.
Worker initiated sub-task size (WIS) strategy

The strategy is based on the semi-decentralized approach of resource management. For 70% of the total task, each worker in HDC system itself suggests its next sub-task size to the master machine while returning the results. However, for remaining 30%, the master reduces the sub-task size of each worker on each worker's request. The strategy is carefully designed so that each worker should not wait to get other workers state information. At the starting of the application, each worker sub-task size is greater than that of RTS. Therefore, the number of request made by the master to distribute the whole task is less as compared to RTS strategy (see Table. 3).

The performance of WIS strategy highly depends upon the performance heterogeneity of the workers in HDC system. If performance heterogeneity of HDC system is lesser, each worker sub-task size is shorter (see equation 3) this causes the increase in number of request made by the client machine to distribute the whole tasks, which may degrade its performance. This strategy is tested on maximum of 20 WSs/PCs. It has average improvement in speedup of 20% to 25% over RTS strategy.

<table>
<thead>
<tr>
<th>HDC system configuration (NM)</th>
<th>RTS</th>
<th>WIS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>640</td>
<td>80</td>
</tr>
<tr>
<td>8</td>
<td>640</td>
<td>88</td>
</tr>
<tr>
<td>12</td>
<td>640</td>
<td>130</td>
</tr>
<tr>
<td>16</td>
<td>640</td>
<td>170</td>
</tr>
<tr>
<td>20</td>
<td>640</td>
<td>225</td>
</tr>
</tbody>
</table>

Table. 3: Comparison of RTS and WIS strategies in term of number of requests made by master machine to distribute the whole tasks.

![Fig. 4: Measured average workers idle time cost (sec) in five HDC system configurations obtained in RTS and WIS strategies.](image-url)
7.4 CONCLUSION

In this chapter we presented our developed adaptive semi-decentralized (WIS strategy) resource management and worker’s self sub-tasking strategy. The results show that the RTS is inefficient for heterogeneous HDC image computing system. The adaptive WIS strategy remedies the defects of RTS strategy. The WIS strategy has 24% approximate speedup improvement over RTS strategy. The workers idle time cost also reduces approximately to 44% with respect to RTS strategy. We believe this proposed strategy would be fine for other distributed applications like, medical imaging, large matrix computation, etc.

7.5 SUMMERY

The main difficulty with cluster computing environment is the continuous changing in the diversities of performance of individual WSs/PCs which requires an effective task partitioning, scheduling, and load balancing to get better performance. In this chapter, we presented a semi-decentralized resources management strategy for a heterogeneous distributed image computing system. The proposed strategy is called semi-decentralized task scheduling strategy for distributed image computing system. This strategy has following main features: i) It adaptively sizes the sub-tasks size for workers according to their performance. ii) It attempts to reduce the inter-process communication time and load imbalance, which occurred in traditional Runtime Task Scheduling (RTS) strategies. Performances of the RTS and WIS strategies are evaluated on manager/master and workers model of HDC system. The measured results of our proposed (WIS) strategy have shown a significant improvement in speedup over RTS strategy.
REFERENCE


Chapter-8

THREADS AND PROCESS PARALLELIZING TECHNIQUES FOR DISTRIBUTED IMAGE COMPUTING SYSTEM

INTRODUCTION

A thread is a semi-process that has its own stack, and executes a given piece of code. Unlike a real process, the thread normally shares its memory with other threads (where as for processes we usually have a different memory area for each one of them). A Thread Group is a set of threads all executing inside the same process. They all share the same memory, and thus can access the same global variables, same heap memory, same set of file descriptors, etc. All these threads execute in parallel (i.e. using time slices, or if the system has several processors, then really in parallel).

Threads programming models offer significant advantages over message-passing programming models. Threads applications can be developed on serial machines and can run on parallel machines without any changes. This ability to migrate programs between diverse architectural platforms is a very significant advantage of threads APIs. It has implications not just for software utilization but also for application development since supercomputer time is often scarce and expensive.

One of the major overheads in programs (both serial and parallel) is the access latency for memory access, I/O, and communication overhead. By allowing multiple threads to execute on the same processor, threads APIs enables this latency to be hidden. In effect, while one thread is waiting for a communication operations, other threads can utilize the CPU, thus masking associated overhead. While writing shared address space parallel programs, a programmer must express concurrency in a way that minimizes overheads of remote interaction and idling. While in many structured applications the task of allocating equal work to processors is easily accomplished, in unstructured and dynamic applications. Threads API's allow the programmer to specify a large number of concurrent tasks and support system level dynamic mapping of tasks to processors with a view to minimizing idling overheads. By providing this support at the system level, threads APIs rid the programmer for the burden of explicit scheduling and load balancing.

When multiple threads are running they will invariably need to communicate with each other in order synchronize their execution. One main benefit of using threads is the ease of using synchronization facilities. Recently, threads have become powerful entities to express parallelism on shared memory multiprocessors and PDP systems. Multithreaded and distributed computing is gaining a wide popularity in the area of high performance computing. In multi-processors/PDP systems, threads are primarily used to simultaneously utilize all the available processors or remote computing resources. The thread-programming model significantly reduces the complexity of the programming on distributed
environments and successfully improves the scalability of the distributed systems [1].

In UNIX operating system, a process consisting of a single address space and a single thread of control within that address space is used to execute a program. Within the process, program execution entails initialization and maintains a great deal of state information. For instance, page tables, swap image, file descriptors, outstanding I/O requests, and saved register values are all kept on per-program and thus per-process basis. The sheer volume of this information makes processes expensive in time and space [2]. In addition, the inherent separation between processes may require a major effort by the programmer to communicate among the processes or to synchronize their actions [3].

In PDP system to control the number of workers/servers simultaneously by the manager/client using the heavyweight technique (e.g. fork () system call), it duplicates the whole process for child processes. Each child process is associated with a worker and responsible to assign task assign and collects results. The child process begins execution on its own memory space and priority. Several processes created in this manner could then communicate via open pipes, shared memory segment, etc. Therefore, each of the inter-process system calls requires an intervention by the kernel, which can be fairly slow as compared with accessing user resources. In fact, the creation of a thread can be up to 50 times faster than the creation of process [4].

A thread executes a portion of a program, cooperates with other threads concurrently and executes within the same address space. Like processes, every thread must have a separate program counter and stack of activation records describing the state of its execution. The per-process information is common for all threads, therefore, it dramatically reduces overhead [5] occurs in heavyweight technique. Threads communicate via library calls, so they are always faster. On uni-processor system, threads are used for overlap I/O operations.

The pthread library provides many good programming features [6] to implement and to enhance the program structure. The speed achieved by any given multithreaded application depends on the thread management and scheduling strategies. Several papers have been published on the importance of lightweight processes creation and synchronization of multithreaded applications [7], [8], [9], [10]. But very less researchers paid attention for practical evaluation of performance implication of threads parallelizing techniques for PDP system. Recently, the famous message-passing package, PVM, for network computing is also switching to lightweight thread [9] paradigm. Vaidy Sunderm [10] strongly suggested adopting lightweight thread technology for future development in heterogeneous programming packages like PVM.

The investigations are typically approached by a manager and workers scheme in which the manager is responsible for tasks distribution and results collection from the workers.
8.1 THREADS & PROCESS PARALLELIZING TECHNIQUES

The detailed descriptions of various threads parallelizing techniques are explained in [9]. Here we explain our developed runtime threads parallelizing techniques. For multi-threads synchronization, the library provides mutex locks, condition variables, read/write locks, and semaphore methods. The mutex locking synchronization is the fastest among various threads synchronization methods [11]. To avoid raising condition between two threads, we used a single mutex variable to synchronize the whole threads scheduling during the PDP application.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>Total number of tasks (640 horizontal scan-lines of the image).</td>
</tr>
<tr>
<td>$i$</td>
<td>Worker machine number.</td>
</tr>
<tr>
<td>$t_i$</td>
<td>Worker’s assigned task.</td>
</tr>
<tr>
<td>$m$</td>
<td>Worker machine.</td>
</tr>
<tr>
<td>$T'_b$</td>
<td>Total number of unprocessed tasks ($T'_b = T - t_i$), and initially $T'_b = T$.</td>
</tr>
</tbody>
</table>

Table 1. Symbols used in the templates of the threads parallelizing techniques -I, II, and III

8.1.1. Threads & process parallelizing technique -I

It is based on the simple paradigm of master-slave. The master (client) launches a set of slave threads, which are equal in number to the remote workers/machines. Each thread assigns a portion of work (one scan-line of the image) to the worker. After assigning task to all the workers, the master should wait for all of slaves to reach a synchronization point or a barrier. After completion of a set of task by all workers, master again creates new set of slave threads and distributes the portion of work from the balance among the workers. This pattern is repeated until all work is done. The phenomenon of master-slaves threads execution is shown in Fig. 1. The thread scheduling and management template of this technique is shown in Fig. 2 and Table 1 contains the definition of the symbols used in techniques I, II, and III templates.

Master  A  B  A  B

Slaves  

Fig. 1 Master-slave execution in threads parallelizing technique -I
8.1.2. Threads and process parallelizing techniques-II and III

The master launches a set of slave threads, which are equal in number to the workers. Each slave thread takes a portion of task from the remaining balance tasks. In technique II, as the slave thread gets result from the associated worker, it exits or dies. After scheduling slave threads, the master continuously checks the existence of all working slave threads associated to the workers. If a slave thread does not exist, the master understands that the slave thread got results from the specified worker and the worker is idle now. Therefore, master creates a new slave thread along a portion of task. This pattern is repeated until all the work/task is finished.

In technique III, the master does not check the validity of existence of the running threads. As a slave gets result form the specified server, the slave thread itself takes a new task from the balance. The task management takes place in slave process. The threads scheduling template of techniques II & III are given in Figs. 3 & 4.

8.1.3. Heavy weight processes multi-tasking/parallelizing technique (HWPPT)

The main process (parent) creates a complete copy of its process (child). Both parent and child execute independently. It is not a fine way to control multiple workers from a single master/client machine [12]. It uses child process to request job on remote workers and waits patiently for their reply. When the workers finish their job and want to return the result to the client machine, the waiting child process then wakes up and dumps the data back into the environment it shared with the parent. HWPPT is used as reference for comparing the performance of threads parallelizing techniques.
Mastiiy
Slave

begin
$T_b = T$;

for worker machine $m_i$ to $m_{last}$
create slave thread $m_i$;

(pixel $t_i$ $m_i$

task allocate to $m_i$;

$T_b = T_b - t_i$;

end for
repeat until $T_b = 0$;

for worker machine $m_i$ to $m_{last}$
if (slave thread not exit)
/
create slave thread $m_i$;

task $t_i$ allocate to $m_i$;

$T_b = T_b - t_i$;
/
end for
end master

Fig. 3 Threads & process scheduling template of technique –II

Master

begin
$T_b = T$;

for worker machine $m_i$ to $m_{last}$
create slave thread $m_i$;
end for
wait for conditional lock (sleep)
end master

Slave

locking
assign task $t_i$ to $m_i$
unlocking
get_result $m_i$ /* thread get results

locking
/* pixel data on x-window system */
unlocking
put pixel data on x-window system

If $T_b = 0$

 locking
assign task $t_i$ to $m_i$

 $T_b = T_b - t_i$;

 unlocking
get_result $m_i$

locking

Put /* pixel data on x-window system */

If $T_b = 0$
8.2 IMPLEMENTATION

The performance of the threads parallelizing techniques were evaluated in terms of, measured speedup, theoretical speedup and workers idle time cost. Here we briefly define these terms.

Power weight:

The power weight of a machine refers to its computing speed relative to the fastest machine in a network of computers [13]. The value of the power weight is less than or equal to 1.

Power weight of a machine

\[ (W_i) = \frac{T_{pf}}{T_{pt}} \]  \hspace{1cm} [1]

where, \(T_{pf}\), is the time taken by the fastest machine in the network to processes the application independently, \(T_{pt}\), is machine’s processing time to processes the application independently and \(W_i\) is \(i\) th machine/worker number. This factor is also used to compute the theoretical speedup of PDP system [13]:

Theoretical speedup of PDP system (TSP) = \( \sum_{i=1}^{W_t} W_i \)  \hspace{1cm} [2]

Where \(W_t\) is the total number of workers/machines in PDP configurations and \(i\) is the specified machine.

PDP system performance heterogeneity

The PDP system's performance heterogeneity can be defined by variance of computing machines power in a PDP system configuration. The most suitable way to quantify the heterogeneity is to use the power weight of the fastest machine
as the comparison reference. Using the power weight of the fastest machine (= 1) [13].

Heterogeneity of PDP system (H) = \sum_{i=1}^{n} 1 - W_i \quad [3]

**Speedup:**

Speedup is used to quantify the performance gain from a parallel computation of an application over its independent computation on a single fastest machine in a heterogeneous PDP system. It is defined as follows:

\[
\text{Speedup of PDP system (SP)} = \frac{T_{\text{ref}}}{T_{\text{pdp}}} \quad [4]
\]

where, \( T_{\text{pdp}} \) is the time taken by the PDP system to process an application.

**Workers idle time cost:**

The activities of worker’s process in a PDP image-processing system is mainly composed of three factors that are:

- Worker setup time to load pattern data file and initialize all programming parameters.
- Worker ray tracing computation time.
- Worker time taken to report result (data) and getting new task from master.

The third factor includes all communications data access delays for start the processing of new task on idle worker. It mainly depends upon the network performance and client machine’s capability to simultaneously parallelize the task on an idle worker.

**Practically we compute this overhead time:**

\[ O_i = (W o r k e r ' s \ p r e v i o u s \ t a s k \ c o m p l e t i o n \ t i m e ) - ( w o r k e r ' s \ n e w \ t a s k \ s t a r t i n g \ t i m e ) \quad [5] \]

The workers idle time cost can be expressed as:

\[
\text{Workers idle time cost} = \sum_{i=1}^{W} O_i \quad [6]
\]

Fig. 5 Raytracing images: a) scene A, b) scene B, scene C and d) scene D
The heterogeneous PDP system used in our investigation is composed of seven Sun workstations (WSs) loaded with SunOS/Solaris, and three PCs (Intel based machines), loaded with Linux operating system; all of these machines are connected to Ethernet. The client machine was used as a manager and worker simultaneously. The network communication is handled by Open Consortium Remote Procedure Call (ONC-RPC) library and XDR filters; pthread library was used to manage the concurrency. The investigation is carried out on a raytracing application because it can be parallelized without complex inter-processor communication. To investigate the performance of threads parallelizing techniques, we used four raytracing images, and where each was composed of 840x640 pixels. The images are shown in Fig. 5. We set five PDP configurations, i.e., the PDP system was composed of number of machine (NM) = 10, 8, 6, 4 & 2. These PDP configurations were configured in such a way that each next PDP configuration was arranged by omitting the two slowest machines from the current configuration.

### 8.3 RESULTS AND DISCUSSION

The runtime task distribution for PDP system until now maintains its superiority on static task distribution at compile time [14],[15]. For this practical investigation, the tasks were distributed at runtime and subtask size was fixed to one horizontal scan-line of the image. All observations were carried out while all machines had no additional load from other users, neither in the form of processes, nor in the form of network load. The high performance heterogeneity (using equation 3) was found in four PDP configurations (NM= 4, 6, 8, and 10) as listed in Table 2. Therefore, the thread synchronization overhead may not effect the performance of the PDP system, which may occur due to locking and unlocking of thread for synchronization and also due to the small sub-task size [14]. If the task size is too larger, it means that the return data from the each worker will be in huge amount and the client may take more time to handle it. While client is taking data from the worker, and if at the same time, another worker wants to return its result, it may have to wait for a long time.

<table>
<thead>
<tr>
<th>PDP system configuration (NM)</th>
<th>Heterogeneity of PDP system (H)</th>
<th>Theoretical speedup of PDP system (TSP)</th>
<th>( T_{\text{pdp}} ) processing time of image scene A (sec)</th>
<th>( T_{\text{pdp}} ) processing time of image scene B (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Tech. 1</td>
<td>Tech. 2</td>
<td>Tech. 3</td>
</tr>
<tr>
<td>2</td>
<td>0.3</td>
<td>1.8</td>
<td>59</td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>1.8</td>
<td>2.6</td>
<td>104</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>3.5</td>
<td>2.85</td>
<td>112</td>
<td>30</td>
</tr>
<tr>
<td>8</td>
<td>5.4</td>
<td>2.97</td>
<td>140</td>
<td>28</td>
</tr>
<tr>
<td>10</td>
<td>7.3</td>
<td>3.1</td>
<td>148</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 2. PDP system computed heterogeneity, speedup, and measured processing time of five configurations

The computed theoretical speedup (TSP) gives the idea about, how much the measured speedups have difference. The comparison of TSP and SP of all PDP configurations and techniques are shown in Figure. 8. It is noted that, using the
computing power of 10 workers (NM=10) we can achieve the maximum speedup of 3.1 times.

All investigation was done on four images, i.e. A, B, C & D images. The time take by the PDP (\( T_{\text{PDP}} \)) system to process image A & B are listed in Table 2. But in this paper, heterogeneity of the PDP system, theoretical speedup of PDP system, worker idle time cost, number of request processed by the client machine as a worker and speedup were measured/calculated from image A only. Because observation behavior of other images for techniques were the same as that of image A. We will mainly discuss our results based on the PDP configuration consisting of (maximum) ten machines, because efficient threads parallelizing have their large impact when PDP system composed of a large number of machines. We will discuss the other lesser workers configuration of PDP system, but it is hard to detect very visible effects.

Threads & process parallelizing technique-I

According to this threads parallelizing technique, the client assigns a unit of task to each worker and then waits for all workers to complete their tasks or in other words, the master waits to join all created slave threads. This means that each worker has to process the equal amount of tasks; however, the examined PDP configurations are composed of unequal performance machines.

Under this technique, for each heterogeneous PDP configuration the PDP system processing time, \( T_{\text{PDP}} \), is highly dependent on the processing capability of the slowest worker in the current PDP configuration. Now let us consider the case when PDP system was composed of ten machines, including the slowest worker of the network, which takes 1370 seconds to resolve the complete image scene “A” (640 scan-line) independently. For the ten workers, each worker has to compute 64 scan-lines of the image; the relative computed time of 64 scan-lines of the slowest worker is \( (1370 \times 64)/640 \) = 137 seconds and measured \( T_{\text{PDP}} \) time is 148 seconds. The difference in processing time (148 - 137 = 9 sec) is due to the non-homogeneous nature of the image, network and file server accesses, etc. The similar facts can be realized for other PDP configurations (see Table 2). From Table 2, it is clear that as the heterogeneity of the PDP configuration as decreases the \( T_{\text{PDP}} \) processing time improving. Consider the case of PDP system composed of two machines, where the computed heterogeneity for this PDP configuration is low and small amount of speedup was measured but for other configurations no speedup takes place, that’s why we have not included this technique in the speedup comparison (Fig. 8) with other techniques. The measured results of this technique conclude that, the degree of parallelizing of the application is inversely proportional to the heterogeneity of the PDP system.

The processing time obtained by this technique under all five PDP configurations is found worst among those of other techniques. Since there is a waiting time of other workers task completion, which effectively make the delay to assign a new task to the idle worker. Therefore, high workers idle time cost is observed in all PDP configurations as compared to other techniques (see Fig. 6). From Fig. 7, we observe that as the number of workers increases, the number of
tasks processed by the client as a worker decreases, because each worker has to compute the equal amount of tasks.

![Diagram](image)

**Fig. 6 Comparison of measured workers idle time cost (sec) of five PDP system configurations obtained by Threads parallelizing techniques I, II, III and HWPPT.**

**Threads & process parallelizing technique-II**

According to the thread scheduling principle of this technique, the master has two duties after initially creating the slave threads:

- To monitor whose associated worker’s slave thread gets return results and exits.
- To create the slave thread associated the idle worker and to assign a portion of task from the remaining task balance.

In this technique, there is a continuous duty of the master to check the existence of the slave thread associated to each worker. The processing time measured by this technique is shown in Table 2. In case of NM = 10, the measured processing time has 82% improvement as compared to technique I. Since for task assignment, there is no need to wait as occurred in technique I, task is assigned to the worker, which becomes idle. The workers idle time cost (from Fig. 6) has 89% improvement w.r.t. Technique-I in case NM= 10 and similarly for other PDP configurations.

The number of tasks processed by the client machine as a worker (server) in PDP configurations composed of workers 4, 6, 8 & 10 are higher as compared to the technique I (see form Fig. 7). This was also due to the task parallelizing mechanism of this technique because as the worker becomes idle, the master assigns the new task. The creation of slave threads at runtime and the master’s continuous checking the validity of the slave threads result in instability and degradation of client machine [16]. That’s way the measured performance of all five PDP configurations never supersedes those of technique -III.
Fig. 7 Comparison of number of scan-lines processed by client as worker in five PDP configurations, obtained by threads & process parallelizing techniques I, II, III and HW PPT.

**Heavy weight processes parallelizing technique**

The drawbacks of UNIX traditional processes parallelizing technique are identified by many researchers [Ronald 1994],[3]. Here our aim is to clearly quantify the performance gain of investigated threads parallelizing techniques w.r.t. this traditional technique.

<table>
<thead>
<tr>
<th>PDP processing time (sec)</th>
<th>Workers idle time cost</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>435% improvement w.r.t. Tech. I</td>
<td>714 % improved w.r.t. Tech. I</td>
<td>40% difference w.r.t. Computed theoretical speedup</td>
</tr>
<tr>
<td>8% degradation w.r.t. Tech. II</td>
<td>14% increases w.r.t. Tech. II</td>
<td>9% less speedup w.r.t. Tech. II</td>
</tr>
<tr>
<td>21% degradation w.r.t. Tech. III</td>
<td>42% increases w.r.t. Tech. III</td>
<td>27% less speedup w.r.t. Tech. III</td>
</tr>
</tbody>
</table>

Table 3. Comparative summary of results for Heavy weight processing technique (NM = 10)

Since the process creation and destroying take lots of computing resources, this makes the client machine heavily loaded, which degrades its capability to assign task immediately to the idle worker. Due to this fact, the number of tasks computed by the client, as worker is less as compared to techniques II & III as shown in Fig. 7. The measured results in Table 2 shows that the scalability of the PDP decreases as the system is expanded with respect to techniques II and III. The comparative detail summary of measured results in terms of processing time, workers idle time cost, and speedup of this techniques with respect to other techniques are listed in Table 3.
From the template of this technique (Fig. 4), once the master creates the slave threads then it goes to sleep (conditional lock) till the full application is completed. It means that after creation of slave threads by the master, the master does not require any intensive processing attention from the computing resources and slave threads have autonomy to take portion of task from the balance and return results as they received from the associated workers. The comparative summary of threads parallelizing technique III with respect to techniques I, II & HWPPT respectively are as follows, where the PDP system is composed of NM = 10:

- The measured processing time is 580%, 18% and 27% and have improvement w.r.t techniques I, II & HWPPT respectively (see Table 2).
- The measured workers idle time cost is 1300%, 50% and 75% and has improvement w.r.t. Techniques I, II & HWPPT respectively (see Fig. 7).
- The measured speedup is 27% and 9% and has improvement w.r.t. Techniques II & IV respectively (see Fig. 8).
- The difference in computed to measured speedup is 11%, 22% and 30% in techniques III, II and HWPPT respectively. From Fig. 8, it is clear that the measured speedup (SP) achieved by technique III is close to the computed speedup (TSP) in all PDP configurations. By comparing the measured processing time, speedup, and worker idle time costs of techniques I, II & III, it provides the evidence that a small change in thread management implicates a large performance effect on PDP system.

The above comparative statements clearly express that the threads parallelizing technique III maintains its superiority over techniques I, II & HWPPT.
8.4 CONCLUSION

In this chapter we studied three threads parallelizing techniques along with remote procedure calls for heterogeneous distributed computing system. It was observed that the efficient thread management had a good impact on the performance of distributed computing system. We proposed the threads parallelizing technique III in which the master launched slave threads and then it went to sleep till the whole application was finished and slave threads had the autonomy to report results and take task from the balance automatically. In this case, each created slave thread was kept busy until whole application was completed. The sleeping of master after creation of slaves threads mechanism save the computing resources of client machine which improves its capability to immediately take results and assign tasks to the idle workers. The proposed technique has a performance improvement in terms of PDP system processing time, workers idle time cost, and speedup with respect to other investigated threads parallelizing techniques and on traditional UNIX processes parallelizing technique.

The lightweight thread has now become a common element in current generation languages. We hope that our investigation will be valid for other languages, which support lightweight thread paradigm. In future, we have planned to improve the performance of the image PDP system by implementing efficient load balancing and adaptive tasks scheduling strategy.

In this chapter we examined the performance of heterogeneous parallel-distributed image processing (PDP) system using three lightweight threads parallelizing techniques. The results are also compared with traditional UNIX parent/child processes multi-tasking technique. For thread based PDP applications with fine-grained parallelism, small differences in thread management are shown to have significant performance impact, often posing a tradeoff between PDP system speedup, workers idle time cost and number of task processes by the client machine as a server/worker. The investigation shows that the thread creation and reutilization give performance gain on thread creation & joining, thread creation & destroying, and traditional UNIX parent/child processes multi-tasking techniques.