White Paper

# Harnessing Multicore Hardware to Accelerate PrimeTime® Static Timing Analysis

December 2009

**Steve Hollands**
R&D Manager, Synopsys

**George Mekhtarian**
PMM, Synopsys

## Why Multicore Analysis?

In the last few years, the trend in CPU performance improvement has shifted from raw computational speed to parallelism as various memory and power (heat dissipation) limits have been reached at the higher clock frequencies. After surpassing clock speeds of 3 GHz, further CPU performance improvement is now achieved by increasing the number of computational elements (cores) within a given CPU package. This means that in order for software applications to run faster on the latest machines, they need to take advantage of the new multicore architecture.

This becomes especially important for electronic design automation (EDA) applications. With growing design size and complexity, continued EDA tool performance improvements are necessary to help design teams meet their schedules. However, given the intricacy of EDA tools, many factors need to be taken into account when implementing multicore analysis hardware support to ensure fast turnaround time and optimal use of compute resources. In this paper, we discuss the evolution of multicore analysis computer hardware; the various approaches to implement multicore analysis support, specifically for static timing analysis tools, such as PrimeTime; and how to optimize hardware and software settings to improve turnaround time.

## Boundaries on Gains From Parallel Computing

There are some basic characteristics in the parallel processing paradigm that fundamentally affect the gain that is achievable. The key factor that determines the achievable gain within a given problem space in the parallel computing domain is the portion of the problem that *can* be parallelized. Let's consider the two components of the problem: the parallel and the serial component.

The relationship between these two aspects of the problem space, the number of CPU cores utilized and the overall achievable gain is governed by Amdahl's law. This law gives us a theoretical bound on the gain as follows:

***Amdahl's law states that***
• *If P* is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and
• If (1 − *P*) is the proportion that cannot be parallelized (remains serial)
• Then the maximum speedup that can be achieved by using *N* cores is

$$\frac{1}{(1-P)+\frac{P}{N}}.$$

The equation shows that as the serial component increases, less gain is achievable overall regardless of the number of cores that are used in the computation. Therefore, controlling the serial component of the problem space is the key to ensuring good results in the multicore analysis domain.

This is illustrated by the chart in Figure 1 that shows an application with a fixed serial component of just 5% and an increasing number of cores (from 1 to 100) used to compute the solution.

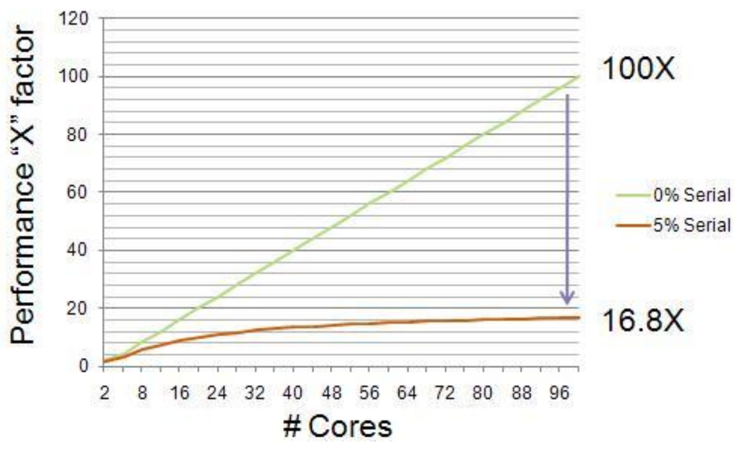

**Figure 1: Impact of 5% Serial Component on Multicore Analysis Performance Improvement**

Clearly, as you increase the core count, the X factor performance gain is severely reduced even with a very small serial component. In this case, a 5% serial component causes over 83% reduction in the achievable X factor flow when 100 cores are utilized.

In the case of static timing analysis tools, the problem space we are focused on is the complete flow. That is the full analysis of a design – from starting the product at the command line to the exit statement at the end of the script.

The serial component in this situation consists of flow and application incurred costs. The flow incurred cost is the portion of the application flow that is not parallelizable. This can, for example, be a set of TCL scripts that are used for specific specialized reporting purposes. Since the execution of these scripts is often not parallelized, they impact the gains achievable from multicore computing.

The application incurred cost is the cost of making a solution parallel. This cost can be further broken down into two components:

- Overhead of parallelization
- Load balancing variation

The overhead of parallelization is the cost, in application software, of making the solution parallel. This serial component cost can be mapped to the "fork" and "join" concepts in UNIX, where in-between, you have the parallel work occurring as illustrated in Figure 2.

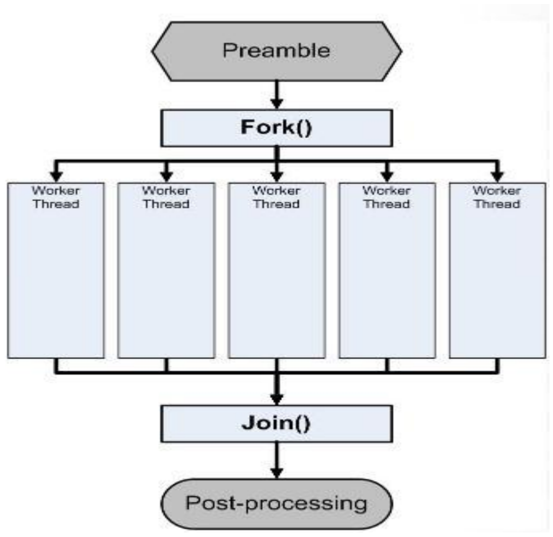

**Figure 2: Fork and Join Concepts in UNIX Illustrate the Overhead of Parallelization**

The load balancing variation is the term used to describe the effect where one worker thread is longer than any other, hence, delaying the "join" process illustrated in Figure 2. This is also frequently referred to as a "long pole".

To maximize the benefits from multicore computing, the tool and flow need to have the following three things:

- Good concurrency model
- Very low overheads
- Good load balancing

Multicore analysis support in PrimeTime was implemented with these things in mind. Significant effort is continually applied to minimize the serial components on the static timing analysis flow and the overhead of parallelization is kept at less than 10%.

## Evolution of Computer Hardware at Semiconductor Design Companies

There are many forms of compute environments deployed at companies today. They range from standalone, single large compute servers with large memory configurations (64/128/256 GB) to farm environments, where many machines are housed and are accessible through farm-based infrastructure management tools. The two most common farm management tools are Load Sharing Facility (LSF) from Platform Computing and Global Resource Director (GRD) from Sun Microsystems.

Over time, computer hardware deployed at semiconductor design firms has been evolving. IT hardware budgets can vary from year to year, but the trend on how this budget is spent has remained consistent. Large enterprises focus on the mass deployment of the lower cost compute hardware while making a few purchases of the latest more expensive machines. This means companies end up with an abundance of machines with 8-32 GB of RAM while machines with larger memory are often scarce and, therefore, more tightly controlled. It is normal for an IT policy to require design teams to make special requests to access such large memory machines.

As the newer machines are all multicore CPU, it is important for static timing analysis tools to provide the flexibility to run optimally on a large number of lower end machines or on a smaller number of higher end machines.

## More Than One Way to Support Multicore CPUs

Applications that run on a single core CPU are referred to as single-core applications. To run on multiple CPU cores in parallel for improved performance, applications need to be developed and architected differently.

There are two basic programming paradigms in the parallel computing domain: threaded and distributed. Each approach provides a unique set of advantages.

The key distinction between these two methods is that threaded multicore parallelism is a single machine solution, where the compute problem is spread across multiple compute cores on the same hardware platform.
Distributed multicore parallelism has the flexibility to utilize heterogeneous hardware platforms, spread over a large computer network or farm. Both methods work to reduce turnaround time, and both methods can be employed on all modern multicore hardware platforms. However, each method uses a fundamentally different set of techniques that bring about certain advantages.

While each technique can be used successfully in isolation to achieve good performance gains, software applications must be able to utilize CPU cores within a single machine as well as CPU cores across the network to get the most from your multicore hardware. This is possible since the two technologies are complementary. Combining the approaches of threading and distributed multicore analysis allows you to harness the gains from both domains.

## Threaded Multicore Analysis

In threaded multicore analysis, many worker threads are established, and each worker thread is assigned a unit of work to complete. Each worker thread operates on a core. The allocation of worker threads to cores is performed by the operating system. Each worker thread is computed separately; however, each thread shares the same memory space. This means that each thread can read from or write to the same location in system memory. As a result, it is essential for threaded multicore implementations to have control mechanisms for such accesses, to prevent data corruption. Threaded multicore analysis is, therefore, referred to as a "shared memory" solution and is focused on activity within the context of a single machine as illustrated in Figure 3. The machine can be standalone or in a farm environment. For example, if in a typical single-core analysis run PrimeTime occupies 100 GB of memory, in the threaded multicore analysis run, the memory footprint increases slightly to 110 GB. A threaded multicore analysis run in PrimeTime can easily be performed on the same class size machine as a single-core run with a typical 2X runtime improvement using four cores.

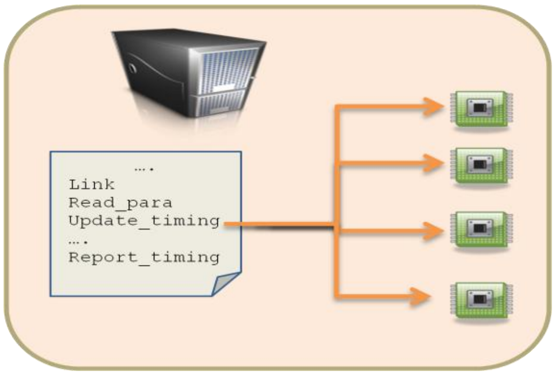

**Figure 3: Threaded Multicore Analysis Focuses on Activity Within the Context of a Single Machine**

One important point to keep in mind when developing a threaded multicore application is the ratio of communication to computation. Worker threads tend to take on small units of computation. If the data communication time is relatively long compared to this computation, the performance benefits from multicore computing are negated. Threaded multicore analysis in PrimeTime is implemented with this ratio in mind by employing mechanisms to control the communication to keep the communication to computation ratio as low as possible.

Furthermore, threaded multicore analysis in PrimeTime is implemented across all aspects of the tool, with threaded operations in the pre-timing update, during timing update, as well as the reporting sections of the flow in order to minimize serial components. In this multicore analysis implementation, significant runtime gains are achieved with a memory footprint that is very close to the single-core analysis implementation.

## Distributed Multicore Analysis

In distributed multicore analysis, the computation is divided into "partitions", which vary by application. For example, test tools can partition test patterns or faults, design rule checking (DRC) or layout versus schematic (LVS) tools can partition rules. For PrimeTime, each partition represents a portion of the design netlist that is transmitted or physically dispatched to a partition process. The partition processes can either run on different cores on the same machine as the master process, or on cores across machines allocated from a farm environment. Figure 4 shows how distributed multicore analysis divides the designs into partitions. Partitioning works best for large designs because the partitioning overhead of parallelization can outweigh the parallel computing performance benefit if the design is small.

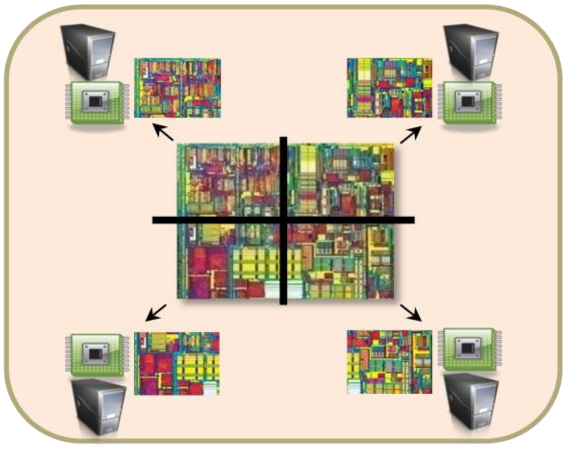

**Figure 4: Distributed Multicore Analysis Divides the Design into Partitions**

The implementation of distributed multicore computing is challenging for a static timing analysis tools like PrimeTime. This is because the partitioning process needs to account for:

- Interdependence of the parasitic-coupled circuits under analysis
- Startpoints and endpoints of all paths in a partition
- Relative partition size and complexity

If the partitioning is not properly balanced or significant, partition-to-partition data communication is required, much of the parallel computing gain can be negated or accuracy can be compromised. In addition, this tricky partitioning step needs to be achieved as quickly as possible to minimize the overhead of parallelization. These are some of the reasons why many other static timing analysis tools have shied away from distributed multicore support. However, all these partitioning challenges have been addressed in PrimeTime while maintaining golden accuracy.

Distributed multicore analysis provides a lot of flexibility. Since each partition represents a portion of the design, the memory consumption of the partition process can be much smaller than the complete design. In addition, the master controlling process does not need all the data that a single-core analysis run would need. As a result, the master and partition processes can run effectively in a farm environment, analyzing large designs that would otherwise require much larger, less accessible hardware. For example, if in a given single-core analysis run, PrimeTime occupies 100 GB of memory, typically you need a 128 GB machine to analyze this design. With distributed multicore analysis, the master requires about 50% of the single-core memory, which is 50 GB in this case. The partition processes require even less memory. Therefore, the distributed multicore analysis in PrimeTime can easily be performed on one class size smaller machines, such as machines with 64 GB of RAM. As discussed earlier, companies typically tend to have an abundance of the smaller class machines.

Of course, a key advantage of using distributed multicore analysis is the significant improvement in performance. On average, you can expect a 2X runtime improvement when using distributed multicore analysis on four CPU cores compared to using single-core analysis.

A further advantage of the distributed environment is that each partition can be executed on whatever hardware is available within the compute environment. For example, in heterogeneous farm compute environments, partitions can be distributed on different computer architectures.

## PrimeTime and Multicore Analysis Support

PrimeTime is uniquely architected to support threaded and distributed multicore analysis. This gives you unmatched flexibility in deploying the solutions in a single machine or across a farm environment.

The distributed multicore analysis capability in PrimeTime helps provide the flexibility to utilize any idle CPU core resource, on any machine in the farm. This enables multicore analysis acceleration with minimum machine access time penalty, which means more timing analysis jobs complete faster and more efficiently. Furthermore, distributed multicore analysis helps reduce memory needs by allowing smaller design partitions to fit in machines one or two memory class size smaller.

The threaded multicore analysis support in PrimeTime can be deployed in situations where high-end standalone machines are available. By enabling multiple cores to read from and write to the same unit of memory, threaded multicore analysis ensures that available cores in the hardware are fully utilized.

Aside from offering great flexibility, the two solutions can work in tandem to multiply the performance benefits and ensure that the full horsepower of your computer resources are harnessed.

To show the benefits of distributed multicore analysis processing, performance results were gathered from real customer designs and are summarized in the following chart:

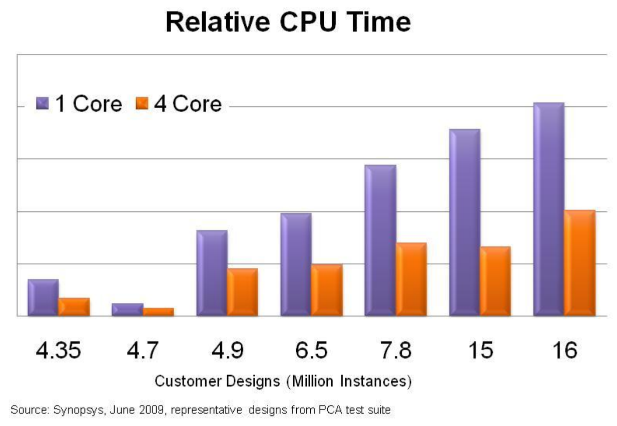

**Figure 5: Runtime Improvement Using Distributed Multicore Analysis With PrimeTime**

The average performance improvement using distributed multicore analysis on four cores is 2X compared to using single-core analysis.

The benefits from threaded multicore analysis processing were also measured using a number of large customer designs. The results are summarized in the following chart:

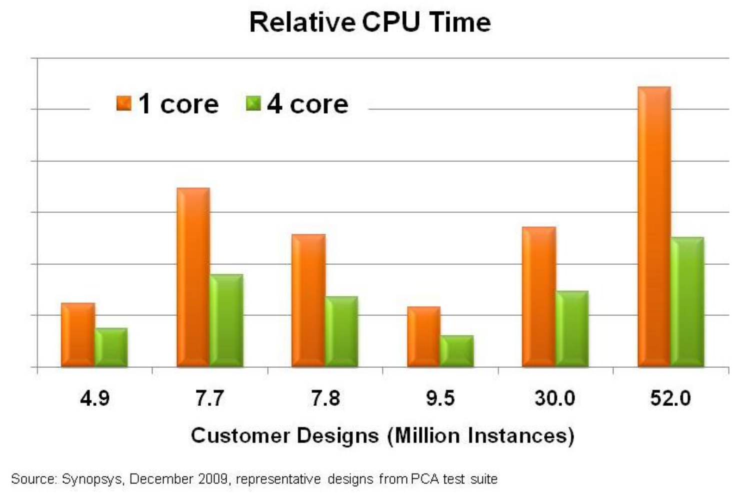

**Figure 6: Runtime Improvement Using Threaded Multicore Analysis With PrimeTime**

The average runtime improvements using threaded multicore analysis on four cores with PrimeTime compared to using single-core analysis are in the 1.8 to 2X range.

As mentioned before, because the two technologies are complementary, they can be used in tandem to deliver further runtime improvements. In theory, the benefit should be multiplicative; therefore, if a design runs 2X faster using distributed multicore analysis and 2X faster using threaded multicore analysis, the combined benefits should be, say around 3.5, which accounts for some overhead. As these multicore analysis solutions get refined and improved, you should be able to see these multiplicative runtime benefits on your designs.

To illustrate the potential of combining the benefits of threaded and distributed multicore analysis processing, experiments were run using a complex 20 million instance design. As the graph in Figure 7 shows, you can obtain up to 6X runtime improvements using this approach.

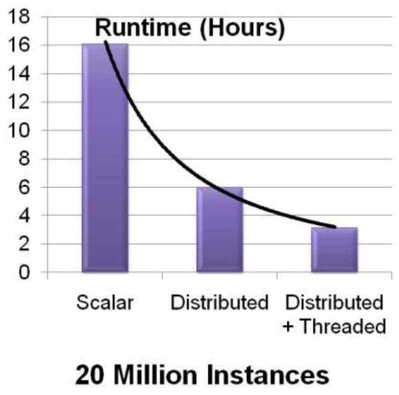

**Figure 7: Dramatic Runtime Improvement When Distributed and Threaded Multicore Analysis are Combined**

## Single-Core and Multicore Analysis Computational Results

In the implementation of multicore analysis solutions for a product such as PrimeTime, which is the industry de-facto standard for timing and signal integrity signoff, we ensure the same high accuracy results as the single-core analysis implementation. However, small differences might exist in the computational result when using multicore analysis compared to using single-core analysis technologies.

In the distributed multicore analysis implementation, for example, you might see slight numerical differences in slack values compared to using single-core analysis because of the partitioning that has occurred on the design data. When the design is partitioned, a new entity is created that contains both the portions of the design (netlist, parasitics etc..) that you are interested in and additional supporting information that is needed to support the partition (in other words, the context of the partition).

As PrimeTime computes slews and delays at every point in the partition, it is working with a smaller subset of the design and can make less conservative assumptions during delay propagation and slew merging. This is similar to how path-based analysis works to reduce pessimism on a specific path. Similar differences result during advanced on-chip-variation (OCV) analysis, clock reconvergence pessimism removal (CRPR) calculation, and signal integrity analysis.

These slight differences in values (less than 1% between multicore and single-core analysis) are expected and are well within the accuracy tolerance of PrimeTime. The final results of stage delay and path delay calculations versus HSPICE simulation are always within the same tight accuracy tolerance whether PrimeTime is run in the single-core or multicore analysis mode.

## Obtaining the Most From Your Farm When Deploying PrimeTime Multicore

When deploying PrimeTime multicore analysis, a well tuned compute environment plays a key role in ensuring the best performance. Semiconductor design firms typically operate a compute environment that is predominantly farm-based. The compute farm houses the majority of the compute infrastructure. This infrastructure is designed to do the heavy lifting for the design teams. Often, there is also older, less powerful hardware in the environment, which for static timing analysis and other compute intensive activities are used for non-intensive tasks, such as farm submission, editing and the like. This section focuses on the farm environment and how to get the most from it when running PrimeTime multicore analysis.

The compute farm and IT infrastructure environment consists of four major components:

- Farm-based computing power
- Hard disk storage
- Network infrastructure
- Farm management methodology

### Farm-Based Computing Power

As discussed earlier, computer hardware at semiconductor design firms typically evolve over time. This evolution is a result of the replacement of older hardware with newer, more up-to-date machines. The turnover pace varies, but, on average, each piece of hardware is replaced roughly after two or more years. This contrasts with the pace of new hardware released from manufacturers. Typically, major vendors, such as Intel and AMD, release new CPU chips on average, around every 18 months. This cycle leads to a highly diverse set of hardware amongst different enterprises. When new hardware purchases are made, managers need to pay attention to the following key characteristics:

- Raw CPU core speed
- Core cache size
- RAM memory size

With multicore computers it is now a well established trend that the raw CPU core speed does not increase significantly. As hardware moves forward, the trend is to have larger numbers of cores within given chips that are housed in sockets on the motherboard.

To determine the best price and performance for static timing analysis, benchmarking of the hardware under consideration is advised. You can do this by using a typical test design, scripts, and methodology.

Consideration should be given to architectures that offer large cache sizes. PrimeTime can make significant use of the cache area to optimize performance. For example, threaded multicore analysis can make use of a larger cache size by accommodating multiple working sets in this high speed access area.

Adequate RAM size is also important. The hardware often utilizes RAM to increase the cache size to meet the demands of the application. In addition, some system memory headroom is required by the hardware and operating system for other tasks. To ensure optimal performance in PrimeTime for single-core, threaded, and distributed multicore analysis, it is recommended that 15-20% of the system memory (RAM) be reserved for such hardware system-based activities.

## Hard Disk Storage

Storage comes in two basic forms: temporary storage, local to the computer, which is also referred to as /tmp and network-based hardware “filers”. For effective multicore analysis, it is important to keep in mind the following characteristics of hard disk storage:

- Device I/O speed
- Number of sockets
- Socket access

During distributed multicore analysis, all the partitions running on remote hosts must have unimpeded access to a common directory. This is known as the “multi_core_working_directory”. Therefore, all farm hosts must have a common view of the disk that this directory resides upon.

The number of sockets on a filer determines how many remote hosts can connect to it at any one time. In addition, the disk I/O speed determines how fast data can be retrieved.

For modern EDA applications, in general, the amount of data to be analyzed is typically very large and covers many aspects, such as basic design information, library data, parasitic information, output log files, and reports. Each of these data components might well consist of tens, if not hundreds of gigabytes. In addition, this data needs to be accessed simultaneously by multiple processes from the farm environment. Clearly, with such large amounts of frequently accessed data, disk I/O is a potential bottleneck to the overall performance of your static timing analysis runs.

It is, therefore, recommended for effective multicore analysis deployment, that input and output data of your static timing analysis runs be hosted on a high speed I/O file storage systems with many free available sockets.

Adequate local storage or /tmp size is also important to ensure optimal performance in PrimeTime. A good rule of thumb is to set /tmp to be at least two times the size of the physical memory on the machine. So, for example, if a machine has 32 GB of RAM, it is advisable that you set the /tmp size on the local disk to at least 64 GB.

## Network Infrastructure

The network infrastructure is another crucial piece that needs to be considered during multicore analysis deployment as it connects your hardware compute platform to the disk data storage. As highlighted previously, the data sizes can be extremely large in modern EDA design and analysis applications. Therefore, the network infrastructure must be capable of supporting these data transport needs.

In distributed multicore analysis, this component of the compute environment becomes vital since the design is distributed across the computing infrastructure. Furthermore, during the analysis, in addition to accessing the disks across the network, each of the partitions must communicate information to and receive data from the master controlling process.

In the case of threaded multicore analysis, as the computation takes place within the context of a single machine, the network infrastructure is less critical, having the same data I/O characteristics of a regular single-core analysis. It is still nonetheless, a vitally important component. Therefore, it is recommended that for multicore analysis, the network have a high communications speed of 1 Gbps or greater.

In addition to the above, the network itself must be reliable. That is the infrastructural components and its connectivity must allow the data to be transported at the maximum traffic rate, and the components of the network, such as compute servers, must be reliable and not fail. Such failures significantly impact the overall performance of your static timing analysis or any other data I/O intensive operation.

## Farm Management Methodology

The compute farm itself is administered by a software management system. There are two main software applications that provide this service (for simplicity, we exclude all others, as the principles and practices are universal):

- LSF (Platform Computing)
- GRD (Sun Microsystems)

LSF and GRD are similar in the functions that they provide, but differ in the terminology that they use for the same operations. While the functional capability of the command set is also similar, the command sets themselves are different.

Generally speaking, within a given enterprise, there usually is an IT department that is responsible for the management of the farm infrastructure. This is the group that has “root” access and determines how the farm is physically configured.

It is recommended, particularly in large enterprises, that there is a direct interface between the IT farm management group and the design engineering environment, such that the needs of the design community can be efficiently communicated and rapid modifications undertaken to facilitate optimal turnaround time and utilization.

It is also strongly recommended that you, as a user of the system, read and understand the user guides for the farm management system that is available to you.

In the following discussion, for simplicity, where a practical example has been given, it is based on the LSF system.

Farm environments are generally set-up with a queuing system. When you submit a job (for example, from your desktop or local machine – referred to as a submit host), the job characteristics are captured in the submission command. The farm management software evaluates the parameters and determines how best to execute the job with the available resources. The submitted job is then placed in the appropriate queue. As the job waits in the queue, resources can be reserved for it. When the job reaches the front of the queue, the farm management software tries to match the requested resources with what it has available at that time. If the resource parameters cannot be met, the job waits in the queue until the request can be satisfied. During this time, other jobs, whose resource requests can be readily satisfied, are dispatched for execution in front of the waiting job. When the resource parameters are finally met, the job is properly dispatched for execution. Once the job execution is completed, it exits freeing up the resources for other jobs. This is a highly simplified description of a complex software application, but it provides a good high-level understanding for the purposes of this paper.

The configuration of the farm can have a significant impact on how well your multicore analysis job executes. Some of the key considerations you should be aware of are:

- How quickly can I get the hardware I need?
- How are the queues set up and managed?
- How are slot allocations controlled?

### *How quickly can I get the hardware I need?*

Obtaining the hardware you need quickly is vital for efficient turnaround time of your static timing analysis. As mentioned previously, companies often have a variety of hardware, which vary in performance and capacity. Generally speaking, there are far fewer larger more powerful (and more expensive) resources than there are smaller ones. Typically, the limiting factor is the system memory of the machine.

As a result, if as you are submitting requests to a queuing mechanism that demands a very large amount of resources from a single machine, such as in a large single-core analysis run, you are less likely to obtain those resources in a relatively short timeframe unless you have a large machine reserved. In some cases, the job waits on a given queue for the resources it needs for long time. If a mistake is made in the request for resources, the job might never leave the queue – the farm management software simply keeps it waiting trying to satisfy an impossible request.

The following charts demonstrate the difference in wait times for different classes of resources:

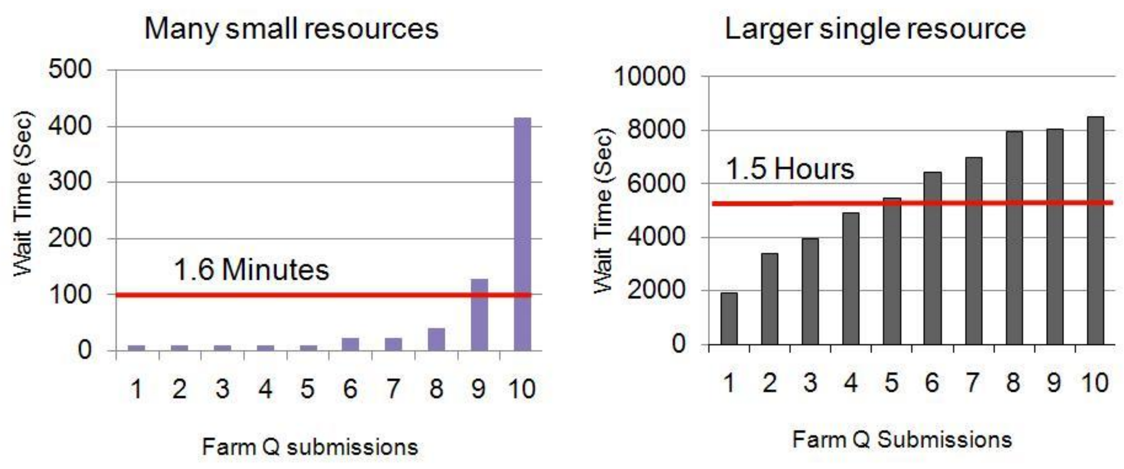

**Figure 8: Larger Resources Typically Have a Much Longer Wait Time**

As you can see, to obtain many smaller resources, the waiting time is, on average, a matter of minutes; whereas, with requests for single larger resources, the wait time can take in the order of hours unless a dedicated resource has been allocated.

It is recommended that you perform test submissions to see how long it actually takes to execute hardware of a particular configuration. This information can then help to determine the multicore approach for the analysis.

In the case of a large memory resource requirement, distributed multicore analysis can be much more efficient than threaded multicore analysis. This is because in distributed multicore analysis you can request a number of smaller memory allocations, rather than one large one.

### *How are the queues set up and managed?*

The way the queue mechanisms are set up and managed is also a vital component of efficient farm usage.
Having a single queue, which operates on a first-come, first-served basis, is the simplest to set up and manage; however, it might not be particularly efficient.

Setting up multiple queues can help to provide a finer grain of control over how and when the resources get used. For example, having a queue that deals with “long” jobs or jobs of a particular architecture are fairly common.

Hardware resources can be associated with, and shared amongst, queues. They can also be allocated to queues upon demand, which is similar to the concept of cloud computing. Having a flexible hardware allocation to the queues can greatly increases the probability of maximized resource utilization, as machines are accessible through multiple entry points to the system, and the characteristics of differing runtime assignments can be accounted for. For example, if there are many small runs (say requiring 0.5-1 GB of memory), it is quite inefficient to delay executing a large job (say one that requires 100 GB), because 50 or so small jobs are running on the larger resource. This can occur simply because the 50 jobs were in front of the large job and the large machine happened to be available. Instead, the smaller jobs should be spread over many smaller machines, thereby allowing the larger job to execute on the appropriate hardware. If there were two separate queues (say "smalljob" and "largejob") with specific machines associated with them, the conflict described would never occur.

There can also be conflict amongst users for position in the queue. For example, a user might submit a 1,000 simulation jobs on a farm that takes many days to complete. During this time, if the queue is first-come, first-served, no other user can execute jobs on this farm, and they all have to wait until these jobs complete and get flushed from the queue.

By utilizing the "fairshare" mechanisms of the farm management software, each user has an allocation that they are allowed to utilize. Once a user exceeds their fairshare limit, the remaining jobs are de-prioritized relative to other users, until their fairshare allocation has been honored. In this way, all farm users get fair access to the resources. This mechanism also supports priority amongst the fairshare, so we can choose to give a higher fairshare priority to one user or group of users. This can be done dynamically to ensure that, for example, compute jobs for a particular project are given high priority in the farm.

It is also possible to allocate certain users to certain queues and, as mentioned previously, for those queues to utilize certain resources. Queues can also be configured for exclusive access to the queue users. Therefore, it is possible to closely and efficiently control how jobs are executed on the farm, using the queuing mechanisms of the farm management software.

### *How are slot allocations controlled?*

The way that slot allocations are controlled is another key area. The "slot" size itself is determined for a given farm environment, and it essentially controls the default amount of resource that is allocated to a job.

Farm-based slots can range from a fraction of a CPU to the entire machine size. If the slot size is set too large, it often leads to resource under-utilization. For example, if the slot size is set to an entire machine with multiple cores and only small jobs are being run in the system, some cores can stay idle. Conversely, if a large compute job is run in a system of fractional CPU slot allocation, the job itself executes sub-optimally, and the job is very likely to be in conflict for resources with other jobs that are allocated to the same machine.

Generally speaking, for single-core EDA jobs, an allocation of one slot per core is the norm. However, in the multicore domain and with the increasing number of cores available, it might make sense for your environment to have an allocation of more than one core per slot per. In the one slot per core allocation, as the number of cores on a machine increases, conflicts arise for other resources, such as memory, on that machine, severely impacting performance. For example, you might allocate one slot per core and have a dual core 8 GB machine as an execution host. In this situation, if the farm manager allocates a 3 GB job to slot #1 and another 3 GB job to slot #2, the two jobs can execute on the same machine without a conflict. However, if the job in slot#2 runs over its 3 GB allocated limit to 6 GB, this causes the machine to swap out to disk in order to allow the host to process the two jobs simultaneously. Clearly, this creates an issue for both jobs that have been allocated to this particular execution host. If the job in slot #1 was performance critical, its elapsed runtime is "artificially" increased.

Well managed farm environments closely control the memory utilization of each job. There are varying levels of control that can be exercised to control memory from simple email notifications to killing off offending jobs. Another alternative is to encourage users to explicitly declare a memory allocation for their jobs by giving jobs without such a declaration a low priority in the queue.

## Conclusion

For the foreseeable future, the trend in CPU performance improvement remains in parallel computing. In order for EDA applications to keep pace with design size and complexity, they need to effectively use multicore hardware. The key to delivering the best performance is having the flexibility to adapt to the evolving hardware resources whether they are high end standalone machines or large compute farms.

PrimeTime has implemented two approaches to multicore support: threaded multicore analysis and distributed multicore analysis. This gives you great flexibility in harnessing the compute power of your hardware whether you are in a standalone or farm-based environment.

In future releases of PrimeTime, the focus will remain on ensuring that the tool can leverage the most up-to-date customer hardware environment. Therefore, the future direction for the product is in the scalability of the combination solution to larger numbers of cores, in both threaded and distributed multicore analysis domains. Distributed multicore analysis will further reduce memory needs by utilizing a larger number of smaller partitions, allowing you to use two to three class size smaller machines and get even faster runtimes.

PrimeTime is getting ready to analyze your “billion instance” design. Are you ready?