

Achieving Optimal Performance Scalability for Physical Verification

By Rahul Kapoor
Marilyn Adan
Louis Schaffer

Introduction

Physical verification runtimes and memory usage have exploded with the increasing number of design rules, their subsequent complexity and the size of chips to be verified. While a traditional approach focuses on speeding up code execution through running multiple threads on multi-CPU (and more expensive) machines, a higher performance to/price ratio can be achieved through an optimal combination of distributed processing and multi-threading. This solution also offers scalable performance which can be maximized through utilization of a network of inexpensive Linux machines. On the other hand, such an approach would have to tackle the issues of data vs. command distribution across machines, network latency, ease of setup and peak memory usage. This article uses real world examples to demonstrate both the need for a scalable solution and some sample performance points delivered by such a solution.

The two most important functions in physical verification are DRC (design rule check) and LVS (layout vs schematic). While a physical verification tool has many more applications, such as layout post processing and rule based optical proximity correction for manufacturing, software runtime always depends heavily on both the size of the layout to be verified (LVS) and number of rules to be checked (DRC)—two parameters that are increasing with every new design. With over 100 million gates at 90nm and the average number of design rules expected to be upwards of 700 (figure 1) at 90nm, it is no surprise that physical verification runtimes on larger designs may be measured in days (sometimes weeks!!) not hours. Despite this, designers are frequently requesting overnight turnaround time (TAT) increasing the need to achieve at least one turn every day or see the results after an overnight run. The demand for fast turnaround is accentuated even more since physical verification is run as a last step before sending the layout to the foundry.

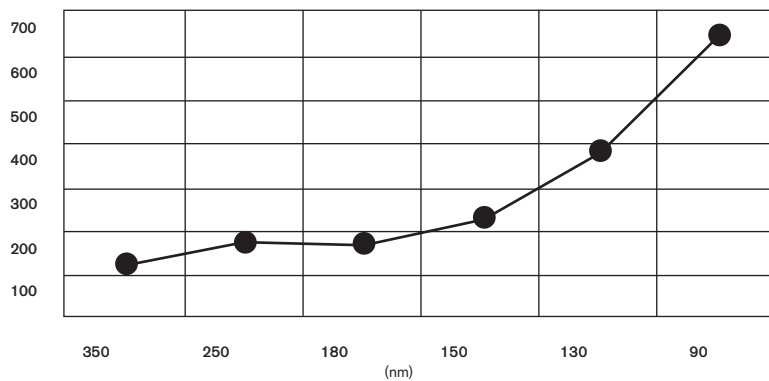


Fig. 1: Quadrupling of design checks from 180nm to 90nm.

Figure 2 shows how the physical verification task can be thought of as several runset commands running on a layout (full chip or block) producing error output and modified layout (in cases where physical verification is used for layout modification as in rule based optical proximity correction). To reduce the turnaround time, an optimal processing platform should be able to determine the runset command and portion of the layout to which the command is applicable, and then send these combinations to separate CPUs for processing. The results can be reassembled at the end.

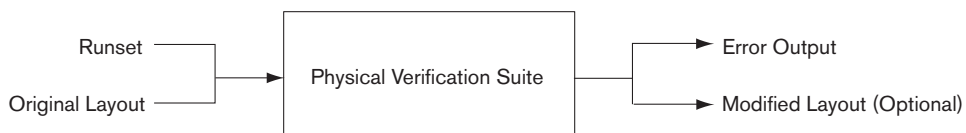


Fig. 2: System view of physical verification problem.

In practice, two approaches have been used to reduce TAT,

- Multithreading allows a single command to be processed in parallel on several CPUs on a common machine. Each of these CPU processes is considered a thread. This is data distribution since a single command is executing on different regions of the layout.
- Distributed processing allows different runset commands to execute in parallel on multiple CPUs. A master process handles command distribution. Each command run has access to the layer data it needs and can run independently on a separate CPU. The separate commands can run across a network on more than one machine or on separate compute engines on the same machine.

Next we consider each approach in more detail.

Distributed Processing

For distributed processing, an advanced scheduler sorts the commands inside a runset and distributes individual commands to separate CPUs. The distribution can be across multiple machines with multiple CPUs or on a single machine with multiple CPUs. The distribution of individual commands requires that each distributed compute engine (usually a CPU) has the capability to independently process separate commands. For example:

A runset might have separate commands for a metal-1 to metal-1 check and metal-2 to metal 2 checks. Since there is no layer dependency, these two category of checks can be performed independently. As one CPU performs the metal-1 to metal-1 check, another CPU can perform the metal-2 to metal-2 check simultaneously. The throughput of the verification run is increased by almost a factor of two. The memory usage of both commands running together is managed by only loading the relevant layer data. It is rare that both the commands achieve peak memory usage simultaneously.

The first challenge for distributed processing is programming the scheduler to organize commands based on input and output layer dependencies. The information about layer dependence is used to schedule when a command will be run. Boolean, spacing, and data preparation commands benefit the most from this layer information. The scheduler also has to track connectivity dependencies to ensure that commands requiring connectivity information are run in the proper order.

The second challenge for distributed processing is to eliminate the “pinch points” in the code. Pinch points are the areas of code which artificially prevent commands from being distributed across the CPUs. As an example, many connectivity commands rely on prior connection of layers in a certain order. This preset connection, or connect database, is used to run one command. The connection can then be enhanced or modified to run another command. Instead of the second command waiting for the first command to finish, the connect database can be automatically replicated by the scheduler across the CPUs so both commands can run in parallel. For example:

The antenna rules for a regular MOS and a high power MOS may be different. However, both devices have the same connectivity through all of the metal layers in the design. A special recognition layer is used to identify the high power MOS device. Otherwise the device layers which create the high power MOS are the same as the regular MOS. Thus replicating the common connectivity for parallel processing on separate CPUs can eliminate the pinch point.

Separate processes (e.g., TCP/IP) are used to communicate between the different components of a distributed run. This allows distributed runs on multiple machines across a network. Excellent utilization (figure 3a) has been achieved running across a network.

| Task Utilized | Running | Idle | Total | File | I/O |
|-----------------------|-----------------|-----------------|-----------------|-----------------|--------------|
| waikiki.synopsys.com | 02:15:29 | 00:02:13 | 02:17:43 | 00:00:00 | 98.4% |
| lua.synopsys.com | 02:00:54 | 00:16:48 | 02:17:43 | 00:00:00 | 87.8% |
| honolulu.synopsys.com | 01:55:35 | 00:22:07 | 02:17:43 | 00:00:00 | 83.9% |
| hana.synopsys.com | 01:55:35 | 00:22:07 | 02:17:43 | 00:00:00 | 83.9% |
| Totals | 08:07:35 | 01:03:17 | 09:10:53 | 00:00:00 | 88.5% |

Fig. 3a: 90% utilization in networked configuration.

Distributed processing depends on the scalability of the runset. The more separate and independent sequences of commands, the more benefit extra CPUs can be expected to provide. Connectivity dependency and layer dependency can limit the number of commands that can be run simultaneously. A good rule of thumb is to use as many CPUs as number of metal levels in the runset. Thus, distributed processing can speedup the verification process significantly. Figure 3b shows the 7x speedup using 10 CPUs for a 90nm TSMC runset. Inexpensive Linux based machines were used across a LAN to achieve this scalability.

| Runtime (hrs) | Number of CPUs |
|---------------|----------------|
| 7.6 | 1 |
| 4.6 | 2 (DP2) |
| 2.4 | 4 (DP4) |
| 2.1 | 6 (DP6) |
| 1.7 | 8 (DP8) |
| 1.1 | 10 (DP10) |

Design and Configuration:

200 million gate
1.5GHz Dual CPU machines
4G to 8G available memory

Fig. 3b: 7x speedup with distributed processing.

Multi-threading

Multithreading enables the execution of a single command simultaneously on different pieces of a layout which increases throughput. The layer data used by a command is cut into multiple regions. These separate regions within a layer can be loaded and processed by separate threads in parallel. For example:

A metal-1 to metal1 spacing check is multi-threaded between two CPUs. A single image of all of the metal 1 in the design is loaded into the machine's memory. Each CPU process, or thread, will perform the spacing check on different regions of the metal layer. The memory required is controlled by the number of regions that the layer has been separated into.

It is useful to think of multithreading breaking the design into regions. The number of regions can vary by design and there are typically more regions than the number of CPUs (each CPU runs one thread.) Larger designs will use more regions to allow more multithreading and better memory management. Multiple threads running at the same time handle the work on these regions. If the number of regions is greater than the number of CPUs, then a thread will start work on the next available region after completing work on a region.

A shared memory approach can constrain the multi-threading performance. Different threads use shared memory to communicate with each other as commercial implementations actively avoid costly disk access. For small to medium designs, a shared memory approach yields fast processing, but very large designs can cause extensive memory swapping to occur.

Combining Multi-threading and Distributed Processing

While both multi-threading and distributed processing are currently in use to achieve overnight runtimes for larger 130nm and 90nm designs today, more efficient hardware architectures are evolving to meet the overnight TAT for future designs.

An important step in this direction is to recognize that distributed processing and multithreading are complementary. Distributed processing can be used for servers across a network while multithreading can be used on multiple CPUs on each server allowing users to get the most out of lower cost 2 CPU servers. Thus two CPU servers can be utilized over the network with two threads each. This combination will take advantage all of four CPUs available. Distributed processing and multithreading may also be run on a single server with multiple CPUs. A 4 CPU server can be effectively utilized by running a 2 CPU distributed run with 2 threads per distributed engine. This will utilize all four CPUs (2 DP X 2 threads) with both the advantages of parallelism at the runset level and at the command level.

As an example of combining DP and MT, consider the how metal-1 and metal-2 checks described above could benefit from the combination of distributed processing and multi-threading:

The first step would be to distribute metal1 and metal2 checking commands between two different dual CPU machines. This is possible because distributed processing can utilize compute resources on different machines. Once distribution is complete, the spacing commands could be multi-threaded between the two different CPUs on each machine. Multi-threading reduces the memory required to execute the single command, while simultaneously boosting performance.

The trend to run combined distributed processing and multi-threading also can be taken advantage of by utilizing inexpensive Linux clusters. The 32 bit Linux servers are cost effective but are limited by the maximum amount of accessible memory. A future challenge is to control peak memory utilization and optimize distributed processing over the network to take advantage of this low cost solution. Currently, 32 bit Linux servers are limited to medium and small designs while higher capacity designs are benefiting from 64 bit servers.

| Runtime (hrs) | Number of CPUs | Design and Configuration: 120 million gate 1.5GHz Dual CPU machines 4G to 8G available memory |
|---------------|---------------------|---|
| 10.5 | 1 | |
| 5.5 | 2 (DP2) | |
| 3.5 | 4 (DP4) | |
| 2.8 | 6 (DP6) | |
| 1.5 | 12 (DP6 MT2) | |

Fig. 4: Multi-threading complements distributed processing.

Figure 4 shows how multi-threading can provide substantial speedup after distributed processing gains start leveling off. In the example, two threads were run on each of the six servers (DP6 MT2) needing 12 CPUs.

Summary

Designers continue to demand overnight turnaround time despite the growing number of DRC and LVS checks and larger data sizes at 90nm and beyond. To meet TAT requirements, a physical verification solution can use distributed processing to exploit runset parallelism and multithreading to exploit data parallelism. Using data generated by Synopsys' Hercules™ Physical Verification Solution (PVS), this paper illustrated how both approaches can be combined to tackle the turnaround challenge for largest chips now and for future processes.

SYNOPSYS®

700 East Middlefield Road, Mountain View, CA 94043 T 650 584 5000 www.synopsys.com

Synopsys and the Synopsys logo are registered trademarks and Hercules is a trademarks of Synopsys, Inc. All other products or service names mentioned herein are trademarks of their respective holders and should be treated as such.

All rights reserved. Printed in the U.S.A.

©2004 Synopsys, Inc. 10/04.PS.WO.04-12648