Parallel Computing and the RTL to GDS Flow

Tom Spyrou
Cadence Design Systems
Distinguished Engineer, IC Digital
EDP 2010

3 Categories of Parallel problems / solutions

- It's the solution not the problem that really defines the category

- Embarrassingly Parallel
  - Compiling RTL modules in parallel
  - Library Characterization of cells in parallel
  - Place and Route of separate Hard Blocks in parallel
  - *Minimal intelligence to divide problem and merge results*

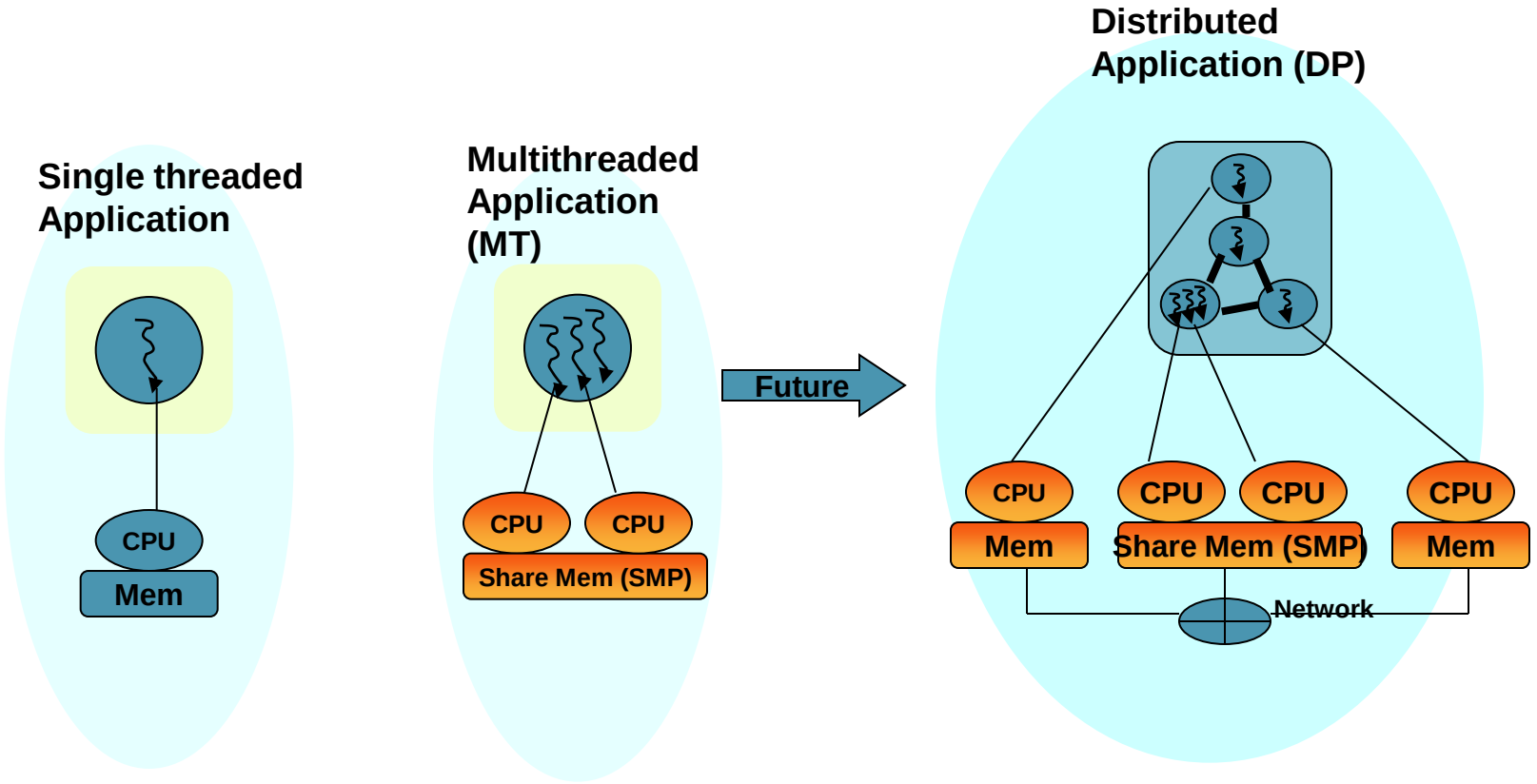
- Course Grained
  - RC Extraction, DRC and problems utilizing rectilinear regions
  - *Some intelligence to divide problem and merge results*

- Fine Grained
  - Static Timing Analysis
  - Routing
  - Physical Optimization
  - *Core Algorithm needs to be coded with parallel processing in mind*

State of the art in EDA

- From the 30,000 foot level and with a wide brush
- The embarrassingly parallel problems
  - Have exploited parallel processing for 1 - 2 decades
- The coarse grained problems
  - Have exploited parallel computing for about 1 decade
  - Some firsts :
    - Calibre DRC, Cadence/Simplex Extraction
    - Synopsys ACS, Magma FineSim
- The fine grained problems
  - This is where the innovation is recent or is needed
  - Interesting things to talk about in other areas but will focus here

Application Parallelization Models



- Easy to develop and debug

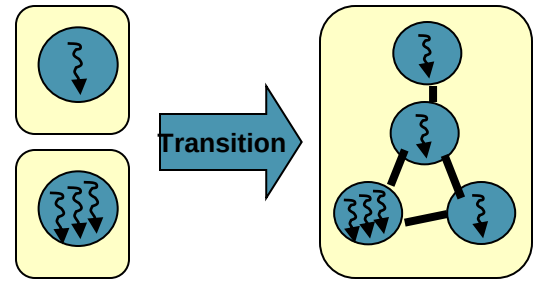
- Shared Memory makes easy to thread/parallelize computations.
- Thread synchronization needed
- Scalability limited by hardware

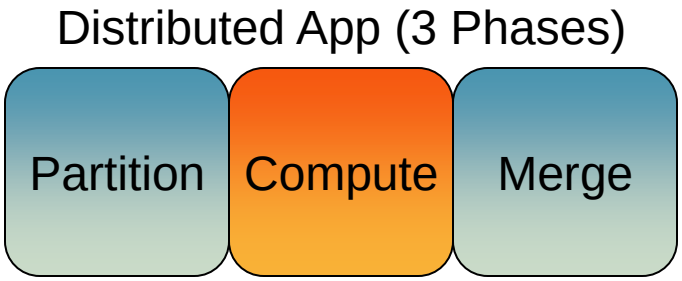
- No shared memory. Connectivity by messages across network backplane
- Requires application re-architecture work to synchronize computations due to lack of shared memory and crossing process boundary
- Excellent scalability, performance, and design throughput

When to use each parallel model

- Single Threaded
  - Non performance critical operations
  - Wish it were still this simple
- Multi Threaded
  - Scalability of problem is lower than cpus in one box
  - Data set is hard to divide even though there is divisible work
  - Very fine grained tasks where sending messages would be prohibitive
- Super Threaded
  - Maximum scalability and flexibility
  - Use of old single cpu legacy machines
  - Data is divisible with problem
  - Time spent in computations is large to offset message cost

Creating a Distributed App



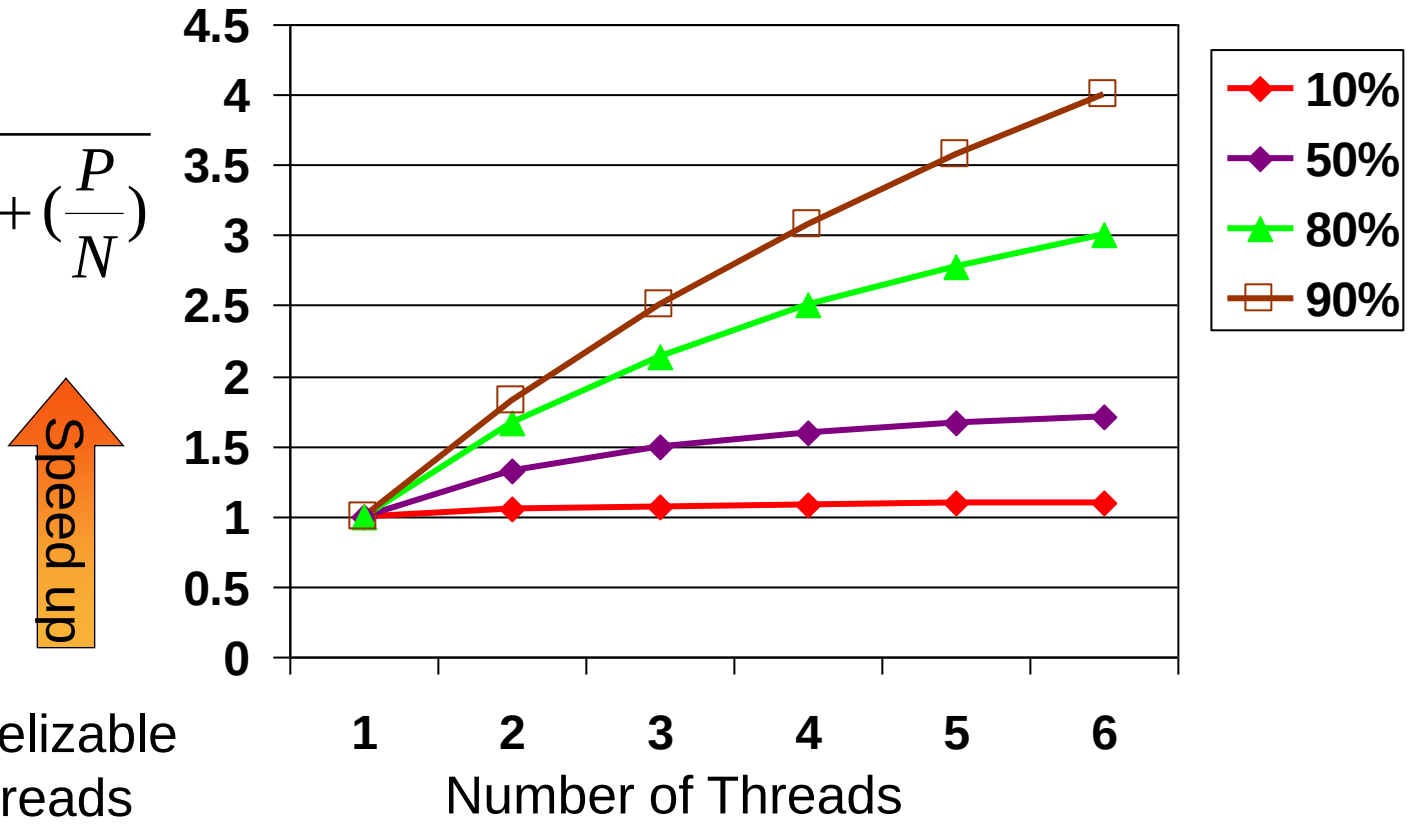


- **Write a distributed task manager**
- **Coordinate computation dependencies**
- **Distributed job control**
- **Optimal use of distributed and multithread "SuperThreads"**
- **Fault tolerance and error recovery**
- **Application Monitoring**
- **Location and platform independent communications**
- **DRM virtualization (independent from lsf, grid)**

Amdahl's Law

"Maximum expected improvement to an overall system when only part of the system is parallelized."

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



P = % parallelizable
N = # of threads

Gustafson's Law (AKA weak scaling)

If (1) the problem size grows as threads are added, and (2) the purely serial component remains fixed, then …

"Any sufficiently large problem can be efficiently parallelized."

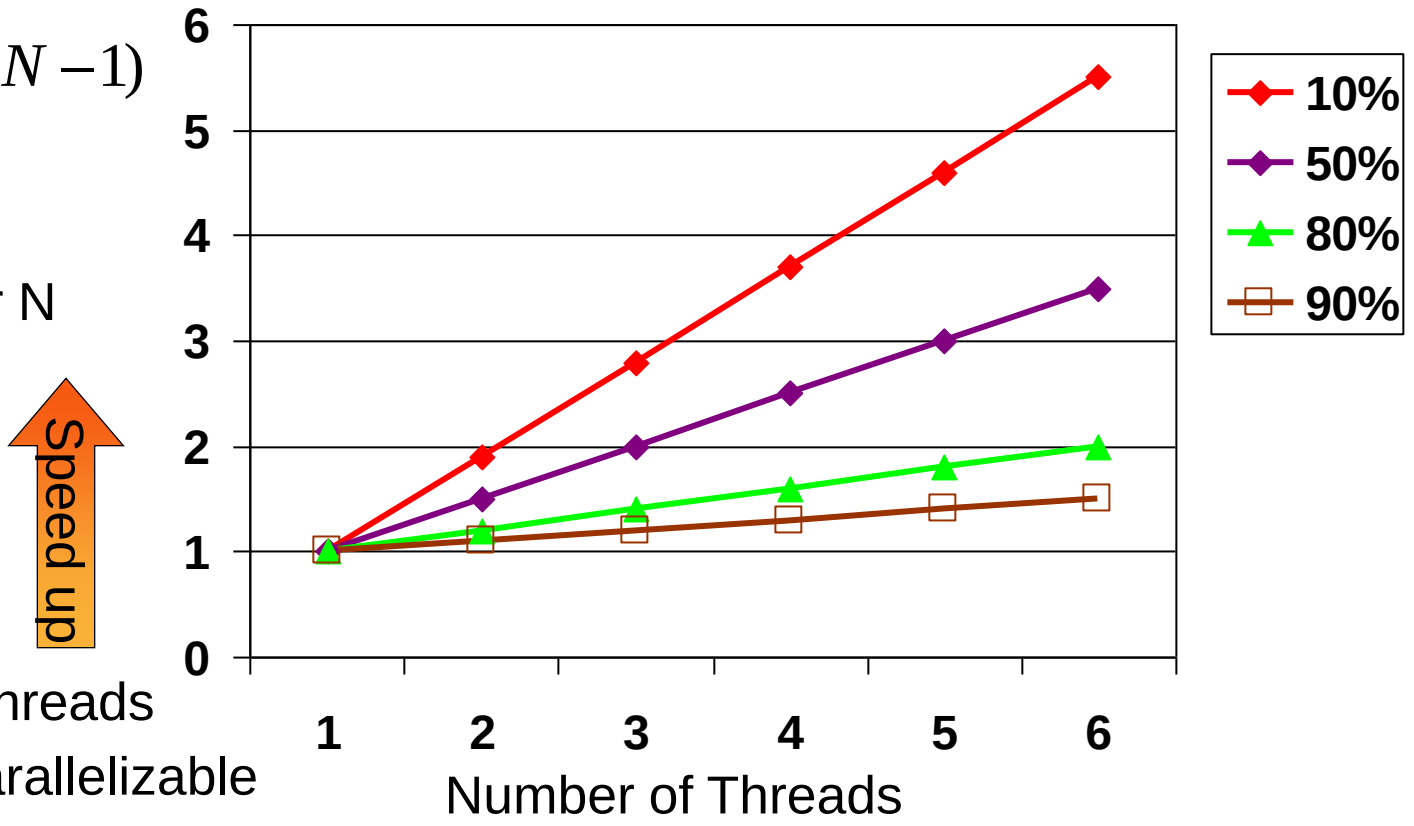
$$S(N) = N - \alpha * (N - 1)$$

$$\alpha = T(1) / T(N)$$

T(N) = time for N threads

N = # of threads

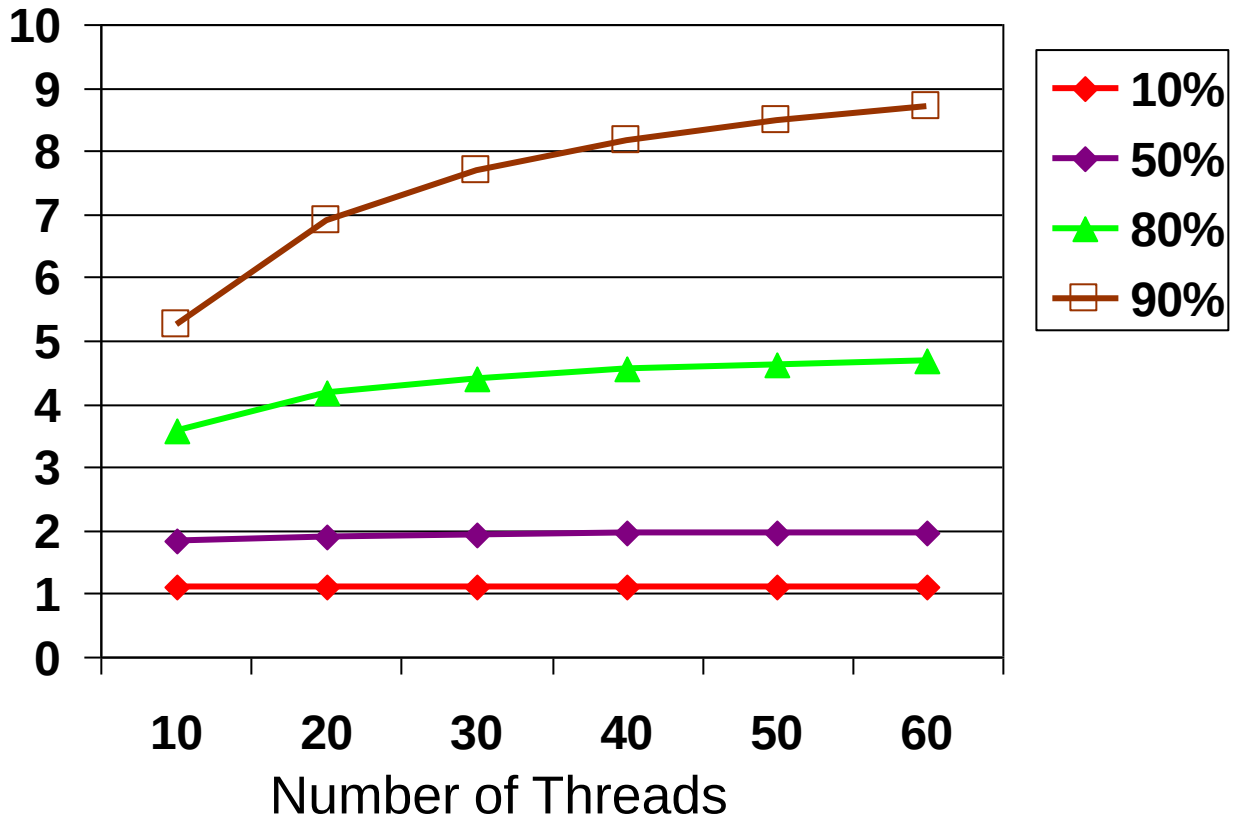
α = % **non**-parallelizable



Amdahl's Law – Future Implications

What happens if we increase to 60 CPUs?

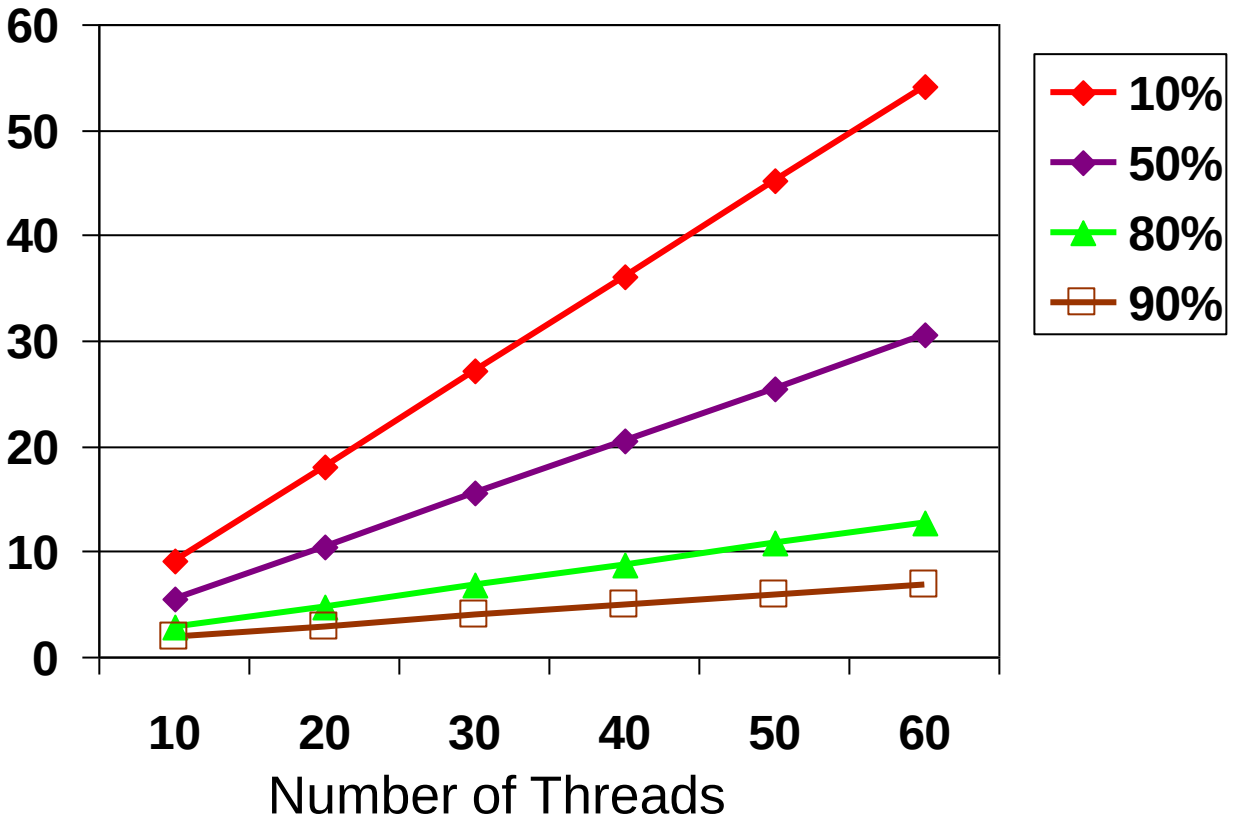
$$\frac{1}{(1-P)+(\frac{P}{N})}$$



Gustafson's Law – Future Implications

Now let's look at 60 CPUs using Gustafson's Law

$$N - \alpha * (N - 1)$$



Result stability with increasing number of Cpus

- EDA users want the same results
  - Adding cpu's is like stepping on the gas
  - Need to avoid torque steer
  - Some small deviations can be tolerated
- Amdahl's law tends to be a better model in this case
  - Take the same app and make it go faster
  - Speedup is based on the percentage of parallelization
- Over time the solution will change
  - Gustafson's Law may take over
- In either case every step needs to be parallelized
  - "End to end parallel processing" will be key

EDA code is complicated
There is a lot of it

- Cadence EDI System example
  - 5M lines of code, most of which was not designed to run in parallel
  - NanoRoute and QRC extraction are notable exceptions

- What if I told my boss we had to rewrite all of it?
  - Good parallel code has to be done from the ground up
  - Hard requirement for Many-Core
    - Cache behavior and data locality
  - For 4-8 Multi-Core we can take some engineering short cuts

- In the long term all of the code needs to be re-worked
  - In the short term how can we make use of parallel processing
  - Focus on time consuming steps
  - Engineering creativity

Example 1 : Parallel Reducer Distributed Solution

- Slaves run in separate processes
- Communicate via TCP/IP
- No threads or thread safety issues
- Good data locality, overhead for data transmission

- Bring up Parallel Framework

- Parallelize calls to the parasitic reducer
  - Absolute worst case example of micro-transactions
  - Small bit of work with significant data transfer
  - Reduction is performed on each net before delay calculation, in any order

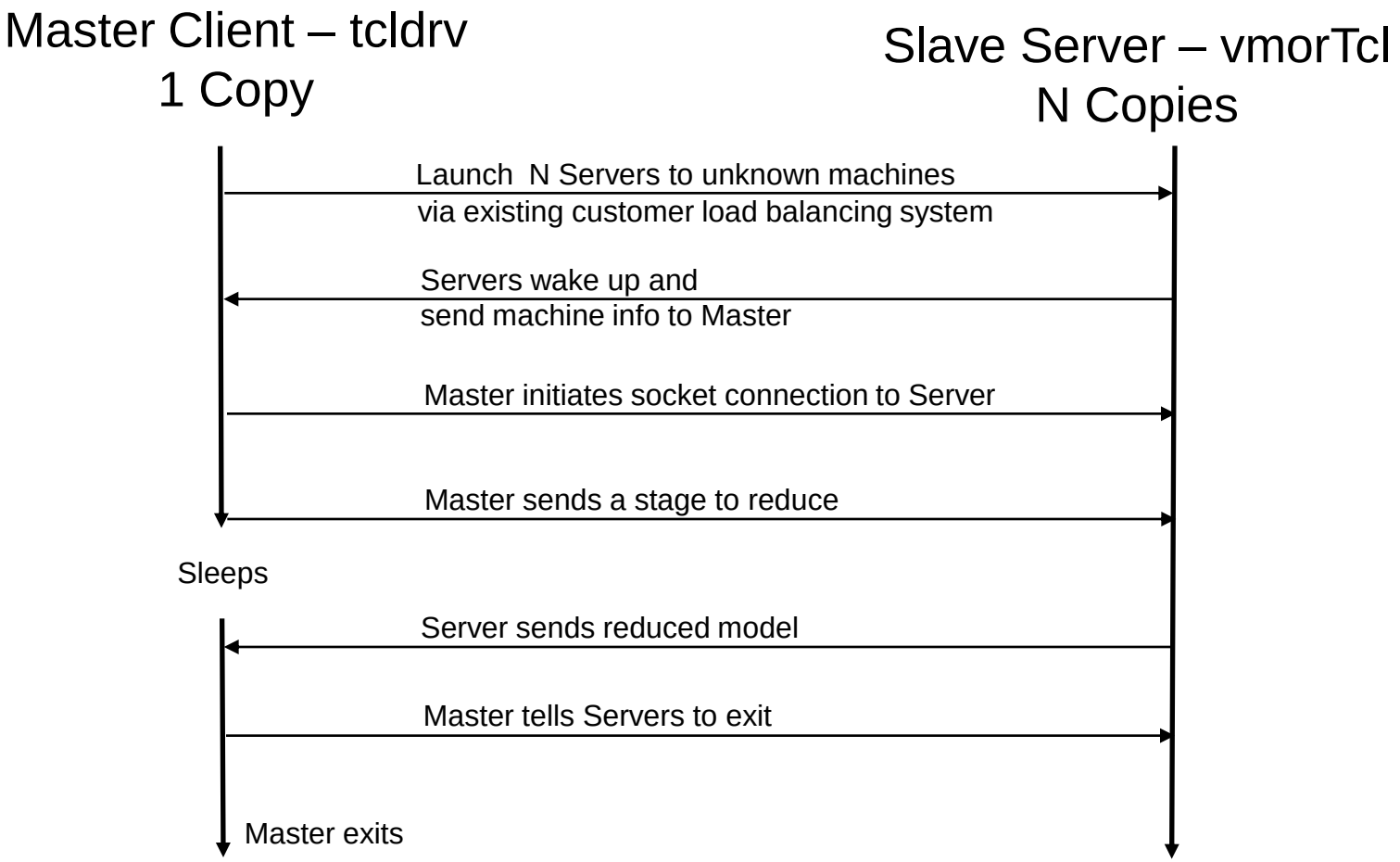
- Use socket based parallelism for micro transactions

- Benchmark Performance

Communication Flow

tcldrv ⟶ EDP using lsf ⟶ vmorTcl

Launches vmorTcl via existing load balancing system



Performance

Number of CPUs	Walltime in seconds
1	1373
2	1596
3	994
4	666

Design has 16182 stages with coupling
Each stage reduced nine times

216 reductions per second using 4 CPUs
4.6 milliseconds per reduction

Simple Architecture

Observations

- Micro-transactions with duration more than 5-10ms can be distributed with reasonable overhead

- Socket based programming is easy to debug if one is careful
  - Make sure the interactions are very simple
  - Try to make the interaction with the server at the tcl command level
  - Server's should be able to be debugged standalone by sourcing a logged set of tcl command messages

- 2K messages of 30KB length per second of bandwidth exists
  - Benchmarked by sending roundtrip messages for 30 seconds
  - 3 GHZ Linux Machine, 1Gbit connection
  - This speed consumes all cpu on master and slave
  - Our case had 216 messages per second of average length 10KB
  - CPU usage of Master was about 8% with 10 cpu's

Observations

- Interrupt driven architecture must be used
  - Master will sleep while waiting for servers
  - Servers will sleep if not busy
  - Polling for messages is very inefficient

- Servers are the slaves, Masters are the client
  - Think of a web browser. It's a client but it is in control of what the servers actually serve.
  - Hence the terminology Master Client and Slave Server

- Side benefit of data locality
  - Each slave builds data structures from the incoming TCP/IP message which are small and local

Distributed compared with Threaded Programming

- Distributed programming advantages
  - Can easily be applied to legacy code which is not thread safe
  - Can scale to farms of cheap machines instead of expensive multi-cpu machines
- Threaded programming advantages
  - Shared memory between threads
  - No overhead to convert complex data structures to string messages
- In the future Many-Core machines locality will be key
  - Which model will work better?

Stateless versus State-full Servers.

- Stateless Server is a server that stores minimal state between transactions
  - Message is received, data structures built, processing is completed, result is returned, memory is de-allocated
  - Our vmorTcl server is stateless and its maximum memory use was about 250MB
  - Just loading library data that won't change is still considered stateless
- Stateful Server is a server that stores state between transactions
  - If the servers load a design and then perform operations on it
  - Synchronization of Master and Slaves can be very challenging
- There is a spectrum of choices, the more stateless the servers, the better
  - Debugging is easier since each message is like a new run
  - Memory use is less allowing use of cheap low memory machines
  - Synchronization of Master and multiple servers is not an issue

Parallel Programs running on farms

- Application should provide a DRM neutral UI
  - Specify queue and machine requirements
  - Specify number of cpus
- Application should never assume it can use all processors on a machine
  - DRM allocates cpus based on resource string
  - DRM does not enforce how many threads an app can start
    - Many threads are for licensing, signal catching etc, low usage
  - If the app takes all the cpus the grid machine will be overloaded

Example 2 : Parallel Noise Analysis

- Must take advantage of multi-core processors
- Very hard to make legacy code thread-safe
- Long term solution: Rewrite all code
- Short-term solution: Copy on Write fork()

What's a Copy on write fork()?

- Process
  - Shares **no** memory with parent
- Thread
  - Shares **all** memory with parent (and other threads)
- Call to fork() with no exec()
  - Shares all memory with parent – but **read-only**
  - If memory is modified, automatically duplicated in child

Fork() without exec()

- In past fork() copied entire process
  - Terrible if going to follow fork() of a large program with exec ls
  - Double the memory is used
  - About 10 years ago fork() started only copying the page tables

- Pages are marked “copy-on-write” (COW)
  - Child and parent share read-only memory
  - Page tables are copied
  - Page-fault scheme copies page about to be modified
  - Much faster than a mutex
  - Avoids most thread-safety issues

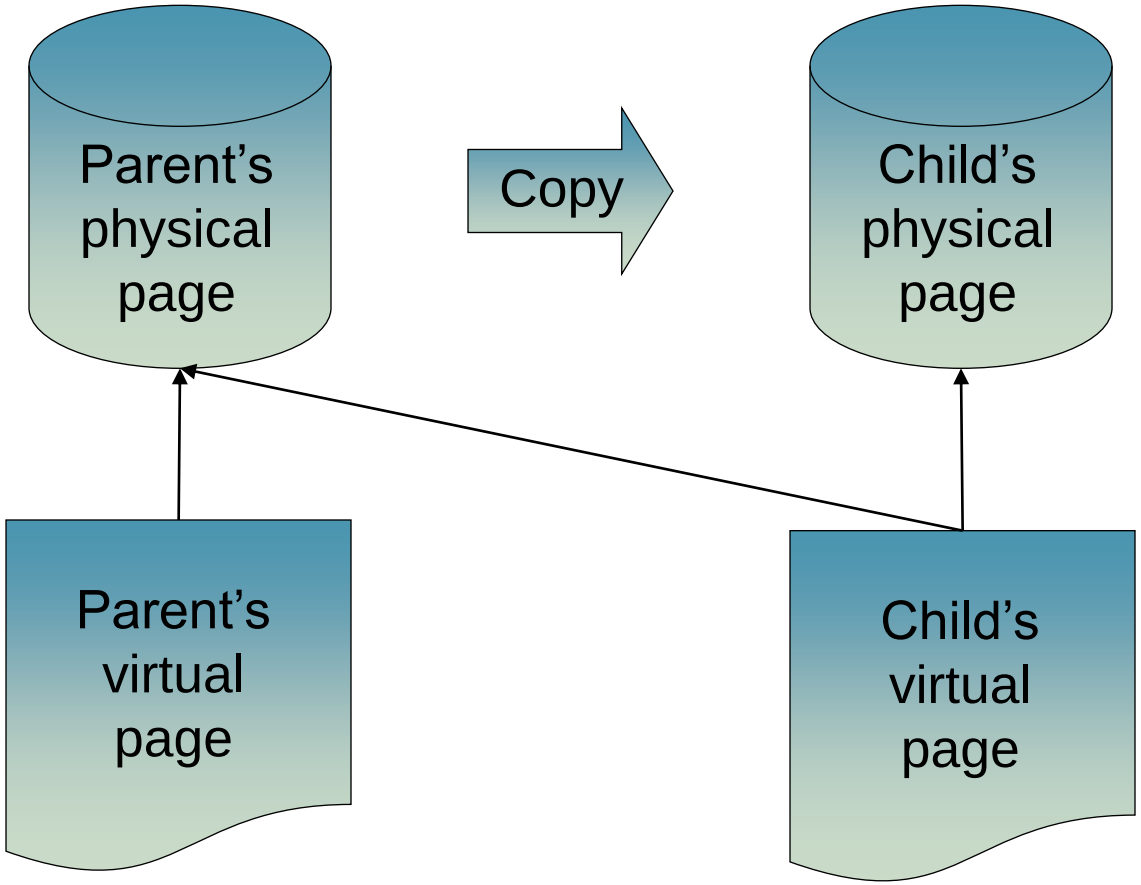
Fork() without exec() code snippet

```
/* Setup communication pipeline first */
if(pipe(commpipe)){
fprintf(stderr,"Pipe error!\n");
exit(1);
}

if( (pid=fork()) == -1){
   fprintf(stderr,"Fork error. Exiting.\n");  /* something went wrong */
   exit(1);
}

if(pid){
   /* A positive (non-negative) PID indicates the parent process */

   dup2(commpipe[1],1);     /* Replace stdout with out side of the pipe */
   close(commpipe[0]);                    /* Close unused side of pipe (in side) */
   setvbuf(stdout,(char*)NULL,_IONBF,0);                  /* Set non-buffered output on stdout */
   wait(&rv);                                             /* Wait for child process to end */
   fprintf(stderr,"Child exited with a %d value\n",rv);
}
else{
   /* A zero PID indicates that this is the child process */
   printf("child process running with id %d\n",getpid());
   printf("child part of parent.c = %d\n", child_test);

   dup2(commpipe[0],0);     /* Replace stdin with the in side of the pipe */
   close(commpipe[1]);                    /* Close unused side of pipe (out side) */

// Useful work goes here
}
```

Copy-on-write

UNIX Fork command starts new process

Data is modified in the child

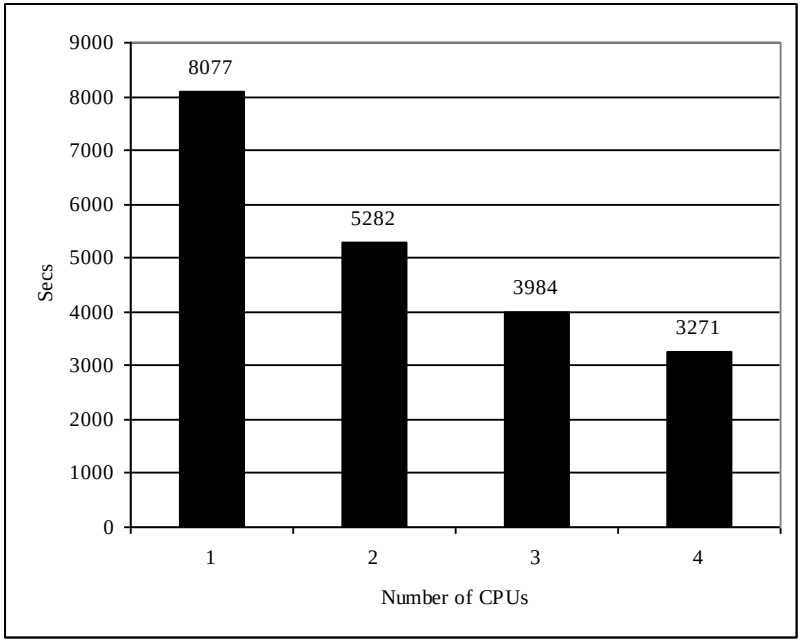


More COW fork () details

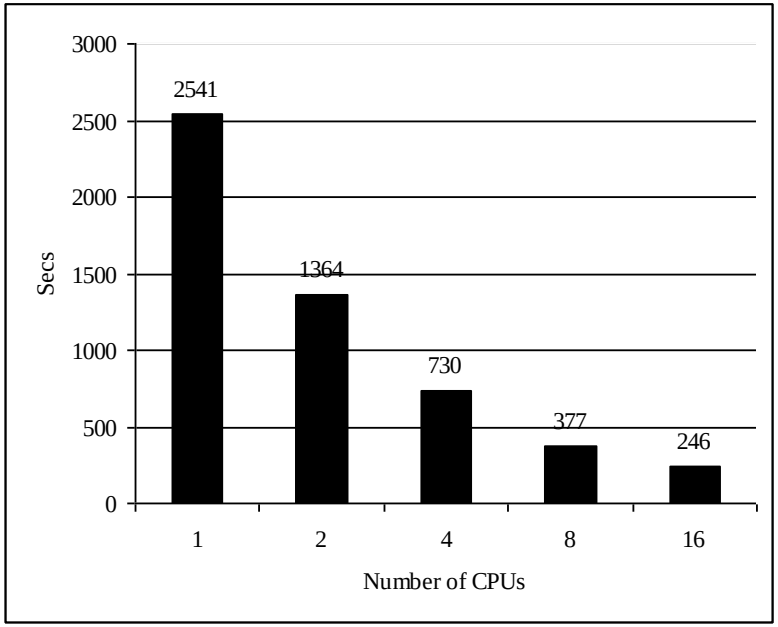
- Ideal for analysis of large data with small output
- Parent MUST do all memory intensive work
- Relatively small additional time/memory cost
  - Time cost depends on size of memory in parent (800ms for 15Gig)
  - About 2% memory overhead
- Number of forks running <= number of cpus available
- Use files/shared memory to return results

Some current results

Analyze noise



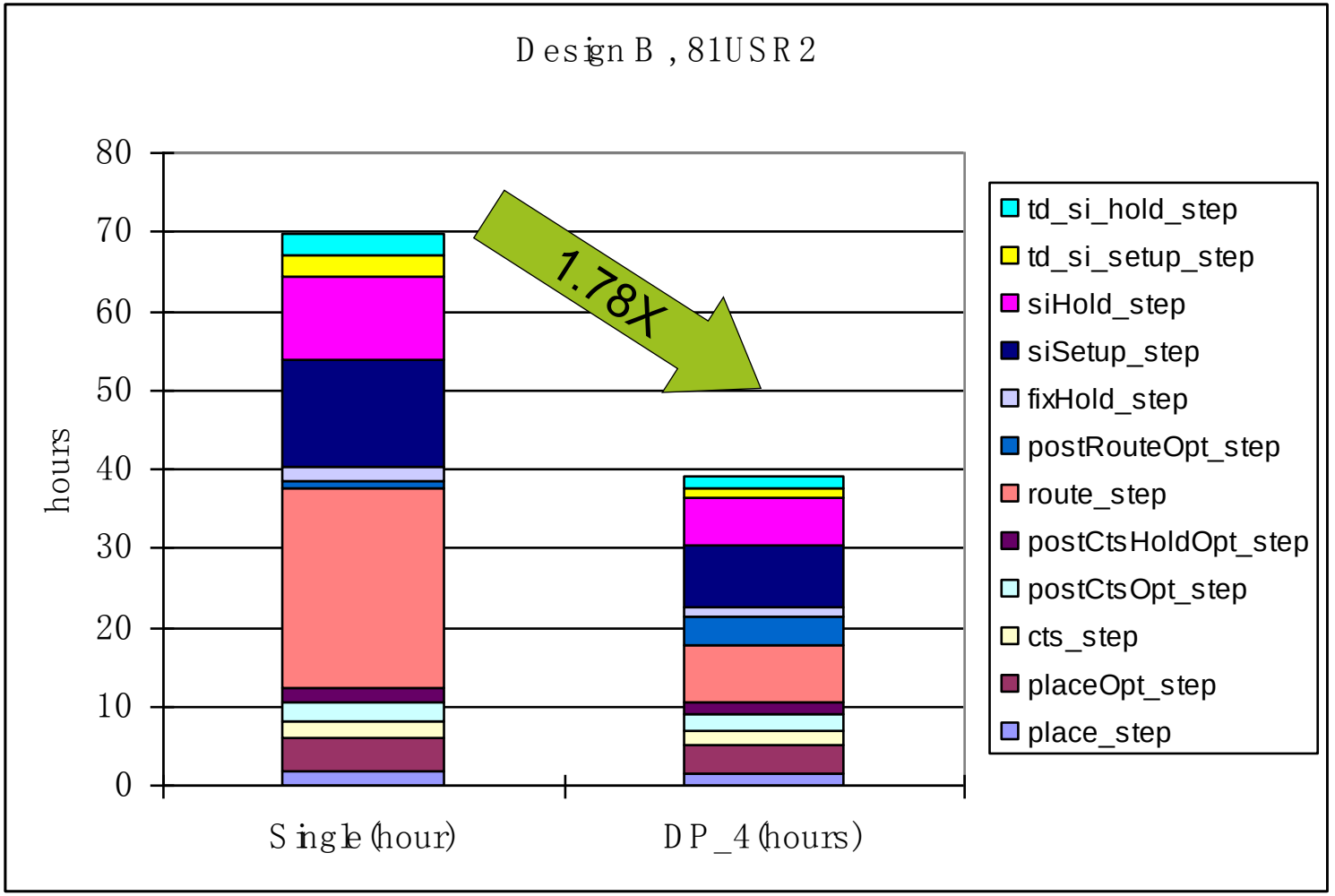
Verify geometry



Conclusion

- Not a substitute for true multi-threading
- Short-term solution for legacy code
- Suitable for many data analysis commands
- Code can be made thread safe over time
  - Move from fork() to pthread when ready
  - Save memory overhead and time for page table copy

Case Study from 2008
EDI System End to End parallel processing
Design : 1M instances, 4 Corners



Fine Grained Parallelism in Core Netlist to GDS EDA algorithms

- Routing
  - Commercial solutions exists
  - Best in class scaled 10-12X on 16 cpus
- Static Timing/Noise Analysis of a single mode/corner pair
  - Commercial solutions exists
  - Best in Class scales 7X on 8cpus for analysis piece
  - One company threads netlist and spef reading gaining more
- Physical Synthesis and Optimization
  - Emerging area, commercial solutions parallelize embedded analysis
- Full flow scalability, netlist to gds, about 2X on 4cpus
  - More possible with certain MMMC designs

Conclusion

- Parallel processing is key to improving performance of EDA applications

- Many companies have implemented
  - Embarrassingly parallel solutions
  - Coarse grained solutions

- A few companies have implemented
  - Fine grained parallelism
  - Multi-Core approaches to similar algorithms

- EDA and how it will work on Many-Core is the challenge
  - When we have 128 cpus, users will expect them to be used
  - Competitors who are behind will lose