Rockwell SOC team

Parallel Distributed Processing on the Flight Analysis and Design System

IEEE Galveston Bay Section Monthly Meeting
October 12, 1995

Mark C. Allman, 244-4031

Goal

- Reduce the execution time of multi-case simulation tasks by distributing the individual trajectory runs across the Flight Analysis and Design System (FADS) workstation network.

- Optimize FADS workstation utilization.

- Minimize the changes to existing software and procedures.

  - Changes to procedures and software involve an expensive testing, validation, and documentation process.

  - Target: no changes to procedures and software.

Background

- Many tasks currently performed by Rockwell Flight Design require multiple simulation runs.

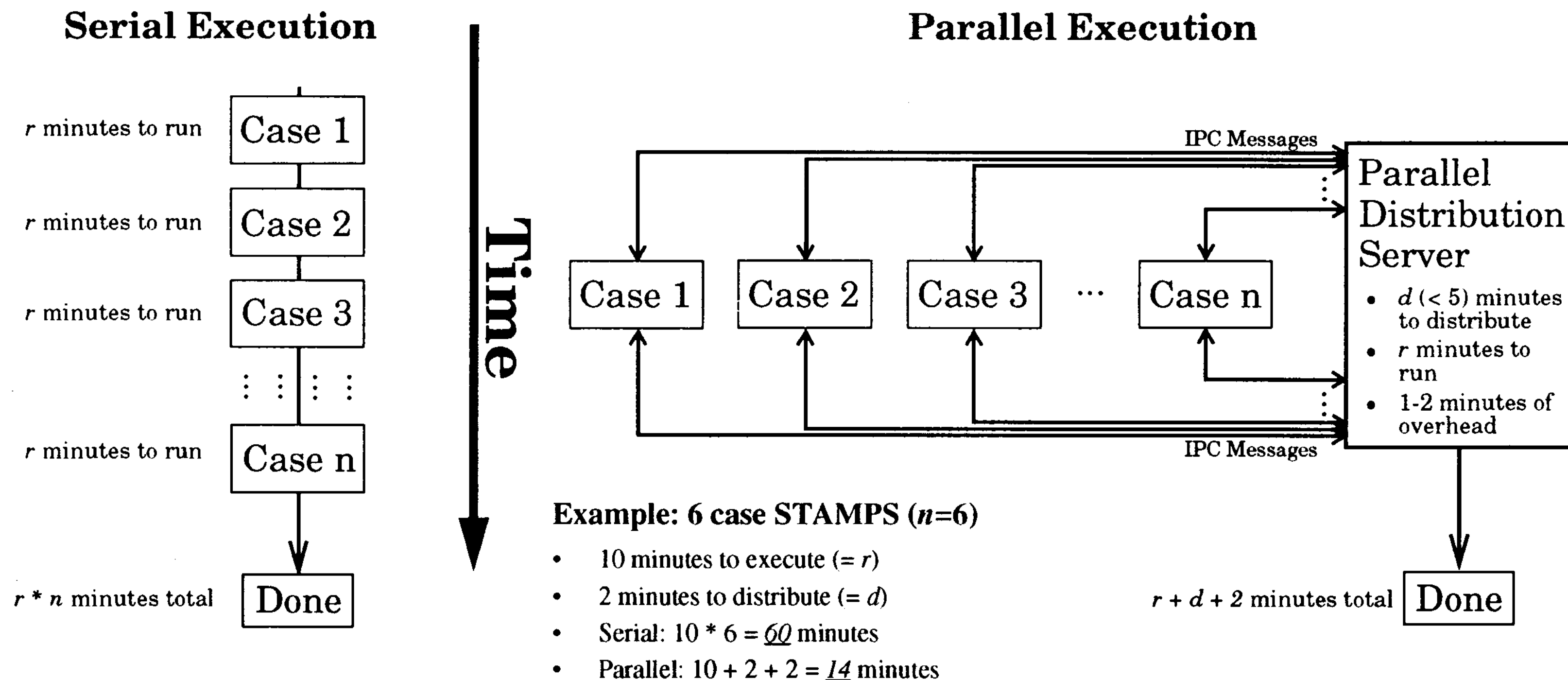
- Each simulation is independent of the others.
  - Later simulations are not modified by earlier results. This is not true for iteration or optimization tasks.

- For multi-case simulations which are mutually independent the potential exists on the FADS network to execute the simulations in parallel, using existing hardware.

- This is referred to as *parallel distributed processing* (PDP).

Simulation Execution using PDP

- Each simulation case is run on a different host instead of all cases run on a single host. Distributing the simulations around the network changes the execution from serial to parallel.

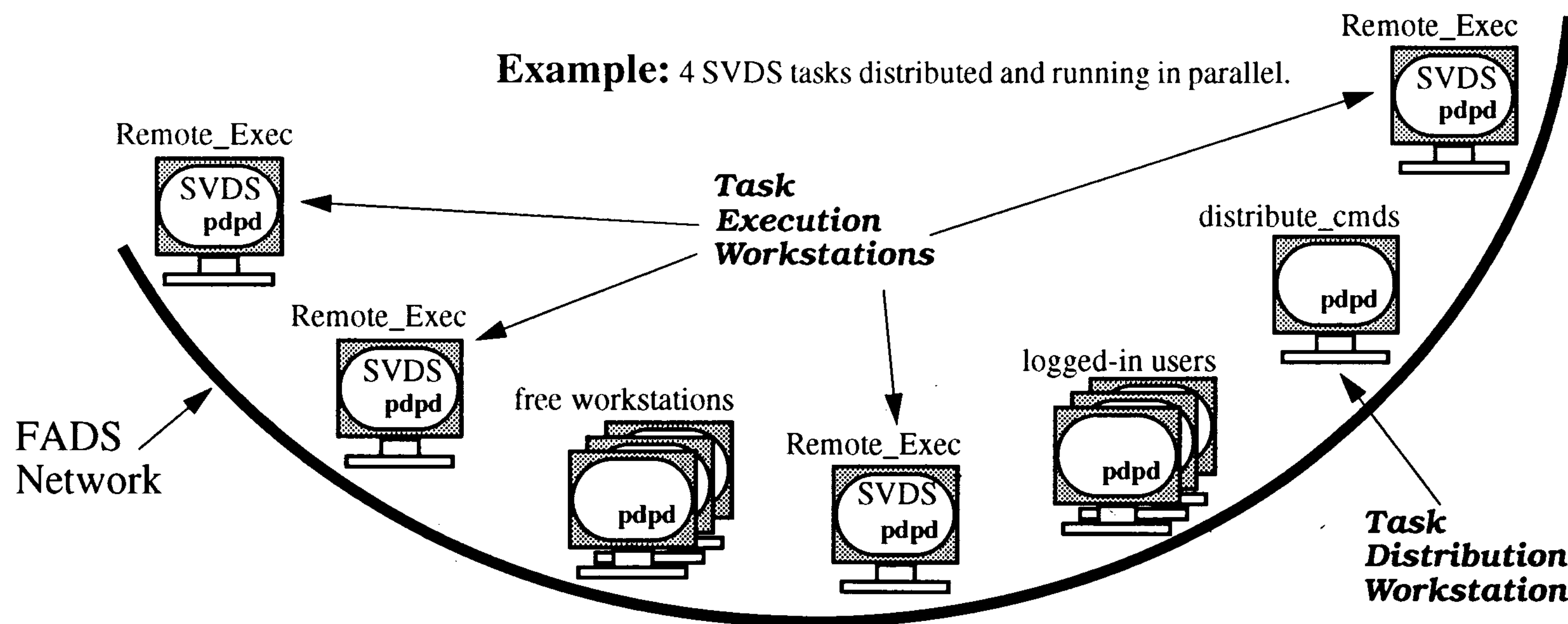


Example: 6 case STAMPS (n=6)

- 10 minutes to execute (= r)
- 2 minutes to distribute (= d)
- Serial: 10 * 6 = _60_ minutes
- Parallel: 10 + 2 + 2 = _14_ minutes

Implementation

- Four programs have been developed to implement PDP.
  - pdpd: Background program which runs on all workstations.
  - getCount: Application interface to pdpd.
  - distribute_cmds: Task distribution manager.
  - Remote_Exec: Remote workstation manager.



Implementation: *pdpd* daemon program

- C-language background program which runs on all workstations.
- Uses Open Network Computing (ONC) remote procedure calling (RPC), registering with the Unix portmapper at start-up and deleting the registration when shut down.
- Written in C to minimize the program size and maximize the execution speed.
- Purpose is to determine the number of Unix processes on a workstation which are owned by a specified user.
    - If the only processes running are system administration/operating system tasks, then the workstation is not being used by an engineer and is available for use by the PDP software.

Implementation: *getCount* interface program

- The getCount program is run from the Unix command line or as a sub-task from a Perl script.
- getCount sends process count requests to the pdpd daemon on remote workstations.
- getCount is also written in C to minimize size and maximize speed.
- Optional *username* argument allow for a one-line determination of the number of user processes on a workstation.

```
getCount chaos
getCount asc1_02 root
getCount chaos -root
```

Total number of processes on workstation chaos.
Number of *root* processes on asc1_02.
Total number of non-*root* processes on chaos.

- Speed: < 1 sec. Using the Unix command *ps* piped to filters: > 15 sec.

Implementation: *distribute_cmds* script

- The distribute_cmds script is written in the Perl scripting language.
- Perl is a fourth-generation language which supports Unix process management (e.g., fork, signal processing) and network communications capabilities (e.g., socket/bind/listen) with a concise syntax similar to C.
- distribute_cmds manages the distribution of user tasks (typically trajectory simulations), keeping track of what tasks are finished, in-progress, and pending.
    - Workstation names currently on the network is obtained using Unix *ypcat hosts*.
- distribute_cmds will wait for all tasks to complete.
    - If the completion status of a task is "killed," then the command is re-submitted to another workstation.

Implementation: *Remote_Exec* script

- The Remote_Exec script is also written in the Perl scripting language.
- distribute_cmds starts Remote_Exec on the remote workstation to execute a single user task.
- The script monitors the user task for completion and the workstation for engineers logging on.
- Remote_Exec aborts the task if an engineer logs onto the workstation.
  - The abort is effected by sending a Unix *int* signal to the user task process group.
- Remote_Exec returns a completion status message back to distribute_cmds:
  - If the user task completes then a "command complete" message is sent back.
  - If the user task is aborted then a "command killed" message is sent back.

Implementation Issues

- Scheduling.
  - Task scheduling entails minimal complexity. Since each user task is assumed to be mutually independent, the main issue is tracking the status of each task.
  - The distribute_cmds script must run as a "server," in that it must continually watch for and respond to incoming messages.
  - The task of identifying free workstations is forked by the distribute_cmds script as a sub-process.
  - The task of identifying idle workstations is a principal bottleneck due to a potential workstation message reply latency. This latency may be caused by, for example, heavy task loading on the remote workstation, heavy network traffic, or a workstation being off-line.
  - The distribute_cmds scripts periodically "pings" each Remote_Exec script to verify that the remote task is still running.

Implementation Issues (Con't.)

- Communications.
  - The pdpd daemon process and the getCount program use ONC RPC to send messages across the network.
  - The distribute_cmds and Remote_Exec scripts communicate using Unix sockets. Each script establishes a server port to listen for messages from the other.
    - While there is one distribute_cmds script running, there can be as many as 200+ remote tasks running. The FADS network has well over 240 workstations.
  - Distribute_cmds communicates with the free workstation identification sub-process using a socketpair, since both processes are running on the same workstation.
  - Only five socket connect requests can be queued. This limitation requires the distribute_cmds script to off-load tasks that would cause delays. This is the reason the free workstation identification task is forked as a sub-process.

Implementation Issues (Con’t.)

- Resource Allocation.
  - The list of workstations returned from the Unix *ypcat hosts* command is filtered to determine the available candidate workstation pool when distribute_cmds is started.
  - Determining which workstations are free and which are busy is a dynamic process, since users continually come and go.
  - The Remote_Exec script is started on a workstation using the Unix *rsh* command. This takes one process slot on the local workstation. Since the PDP software must handle upwards of 200+ remote tasks, this can overflow the maximum number of processes allowed (typically around 100).
  - After the Remote_Exec script is up and running on the remote workstation, the local *rsh* process is no longer needed. The distribute_cmds script keeps track of what processes are created, and deletes the *rsh* processes that are no longer needed.
    - The process table overhead is on the order of 3 process slots, independent of the number of tasks being managed.

Usage

- Using the PDP software requires no modification to existing simulation software.

- The tasks to be run in parallel are listed in a file, the format being *exactly as they would be typed at the Unix command line*.
  - Each line in the command list file is a task to be distributed.

- distribute_cmds can be executed at the command line, in background, or in batch.

- Output from each command not re-directed is captured in a log file named for the workstation.
  - Example: Output from a command submitted to workstation chaos is captured in the log file *chaos.log*.

Results: Performance Enhancements Project

- Design of 18,000 I-load sets for Performance Enhancements certification.
  - 150 I-load sets for each month varying the measured wind.
  - High, mid, and low dynamic pressure ("Q") cases.
  - Inclinations 28.45, 51.6, and 57 degrees.

- Payback using <u>prototype</u> PDP software.
  - Execution time for each set of designed I-loads is 30 minutes.
  - ~120 different cases at 150 I-load sets per case.
  - Serial execution: 120 * 150 * 30 = 540,000 minutes = ***375*** days to complete.
  - Parallel execution: 120 * 140 = 16,800 minutes = ***11.6*** days to complete.
    - Prototype software required ~140 minutes to distribute and execute 150 simulations.
    - Estimate for the current software release is ~75 minutes.
  - Rough total calendar time savings estimate: 40+ months down to 3 months.

Conclusions

- Parallel distributed processing increases productivity by utilizing existing resources to reduce multi-case task completion time.
  - A potential increase in compute capability (relative to current usage) of 2500%.
  - No additional hardware needed.
- The tasks run in parallel must be mutually independent.
  - Inter-dependent tasks require development of parallel processing software.
- Rockwell Flight Design currently has many tasks that can be improved using PDP.
- No change to current simulation software or design procedures.
- Logged-in users are not impacted since tasks are only distributed to idle workstations.