



# Dakota Software Training

Parallelism

<http://dakota.sandia.gov>



*Exceptional  
service  
in the  
national  
interest*



Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

# Module Goals



- Discuss what to consider when designing a parallelized study
- Understand what Dakota provides and its limitations
- Be able to choose the best parallelism approach
- Know how to configure Dakota and your interface for your parallelism approach

# Opportunities for parallelization



## Example 1: Parallel simulation

- The user's simulation code has been parallelized using MPI, OpenMP, GPU, etc.

## Example 2: Gradient-based optimization

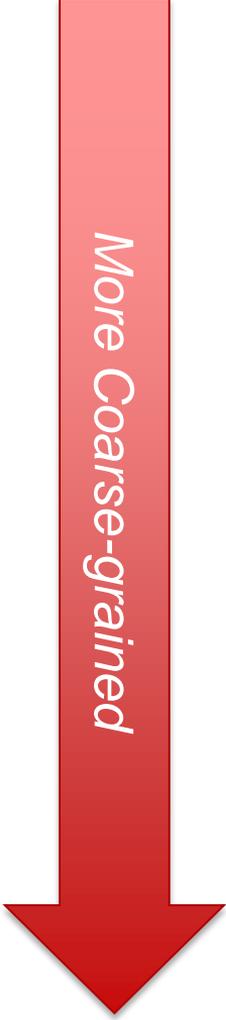
- Finite differencing can be performed in parallel

## Example 3: Sampling

- Every sample is independent of all the others

## Example 4: Multi-start optimization

- Every optimization is independent of all the others



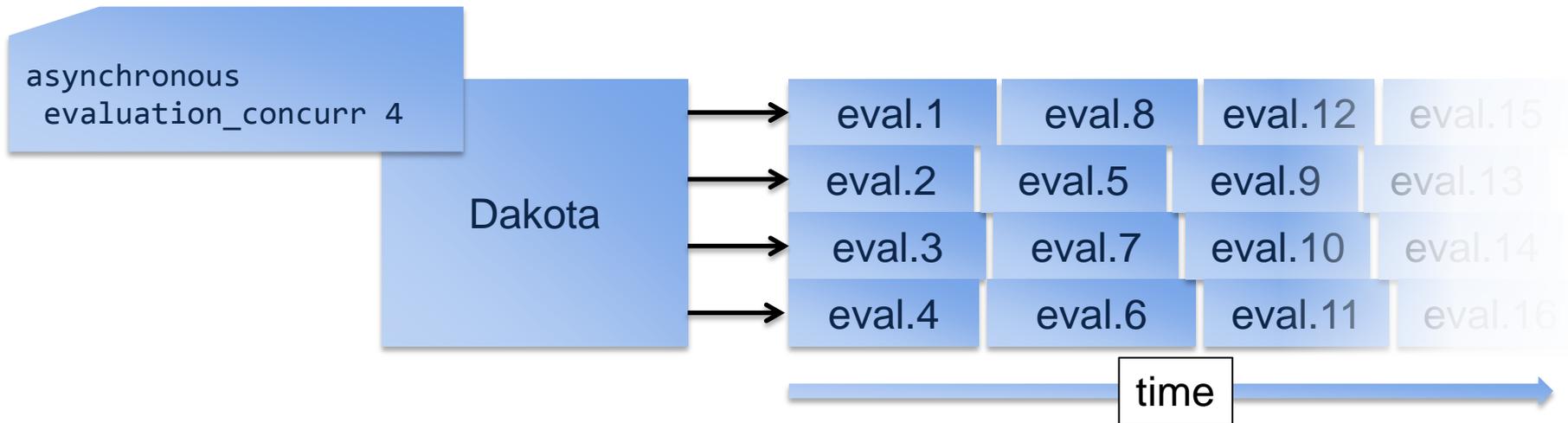
More Coarse-grained

# Things to Consider



- **Available Concurrency**
  - Adaptive vs. single pass algorithms
- **Characteristics of your simulation**
  - Serial or parallel
  - Parallel scaling/efficiency
  - Memory requirements
  - Duration
- **Characteristics of computing resource**
  - Number of cores and memory
  - Time limits
  - On some HPCs, “fork” and “system” are disallowed

# Local Parallelism



One instance of Dakota launches multiple instances of the analysis driver

- Simple and portable
- Works with either serial or parallel simulation codes
- Method of choice for desktop computing

- Evaluations will not be launched across a network (Hence “local”)
- Iterators run sequentially

# Serial versus Parallel Simulation



- Suppose your simulation has been parallelized and your workstation has **24** cores.
- Naturally, you want to use all of them and minimize how long your Dakota study will take.
- **Which combination is best?**

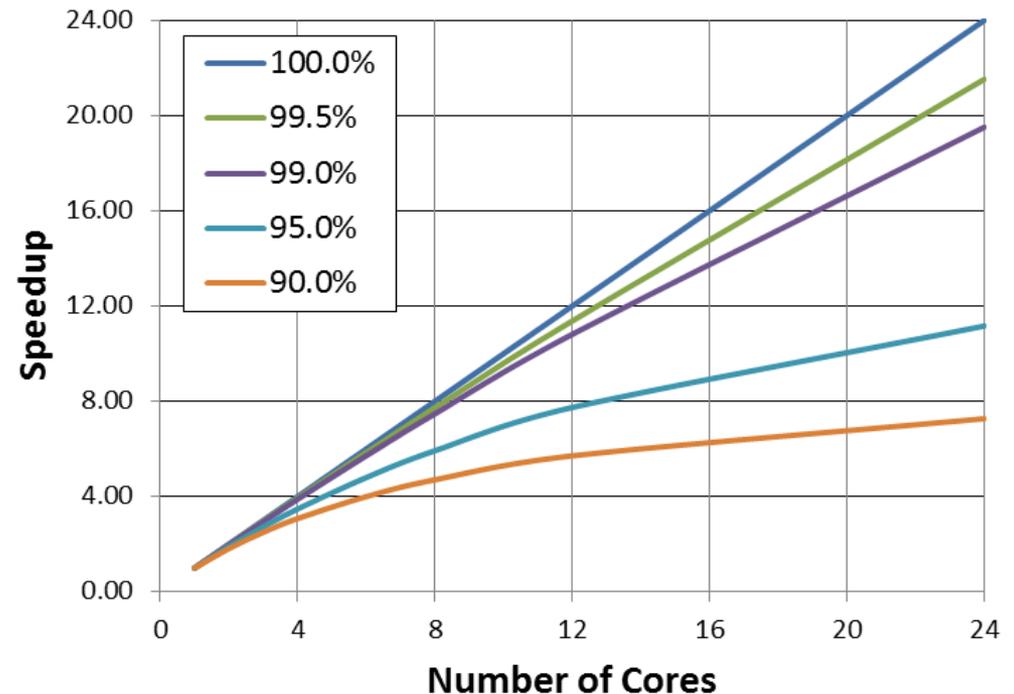
Evaluation Concurrency	Cores per Evaluation
1	24
2	12
3	8
4	6
6	4
8	3
12	2
24	1

# Serial versus Parallel Simulation



- **Parallel efficiency**
  - Fewer cores are better
- **Memory requirements**
  - Upper limit on number of concurrent evaluations
- **Available Concurrency**
  - Another upper limit on number of concurrent evaluations

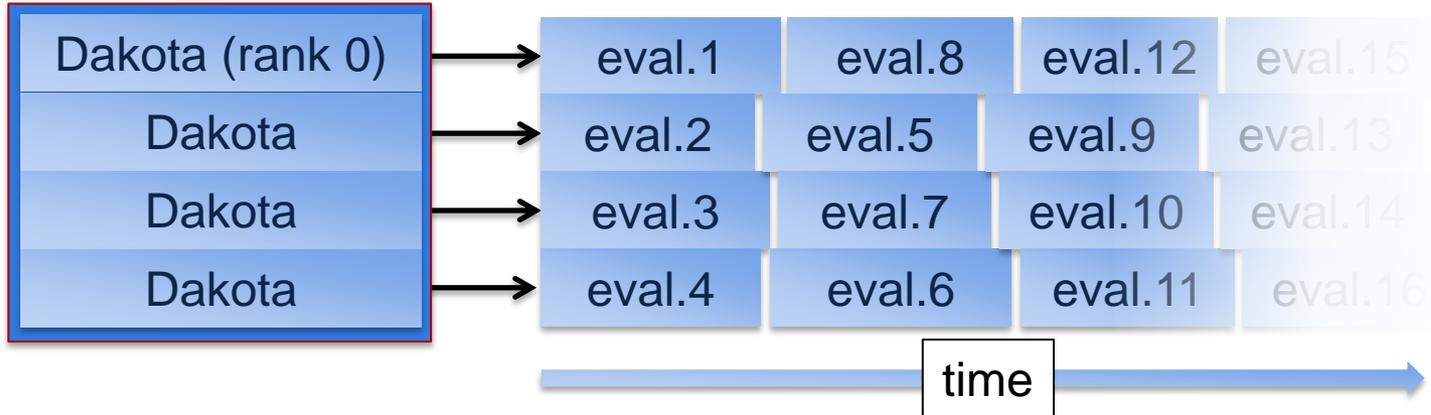
Amdahl's Law



# Parallel Dakota



```
$ mpirun -np 4 dakota my.in
```



Dakota launched in parallel; each “rank” runs analysis drivers

- Still pretty simple..
- Works across the network
- Parallel iterators (**experimental**)
- Dakota highly configurable

- Serial simulations ONLY
- Not supported on Windows
- Dakota must be built with MPI support
- Dakota highly configurable

# Dakota, “Large” Simulations, and HPC



How can Dakota manage evaluations that require large\*, parallel simulations on many cores?

\*More than will fit on a workstation



Two strategies—

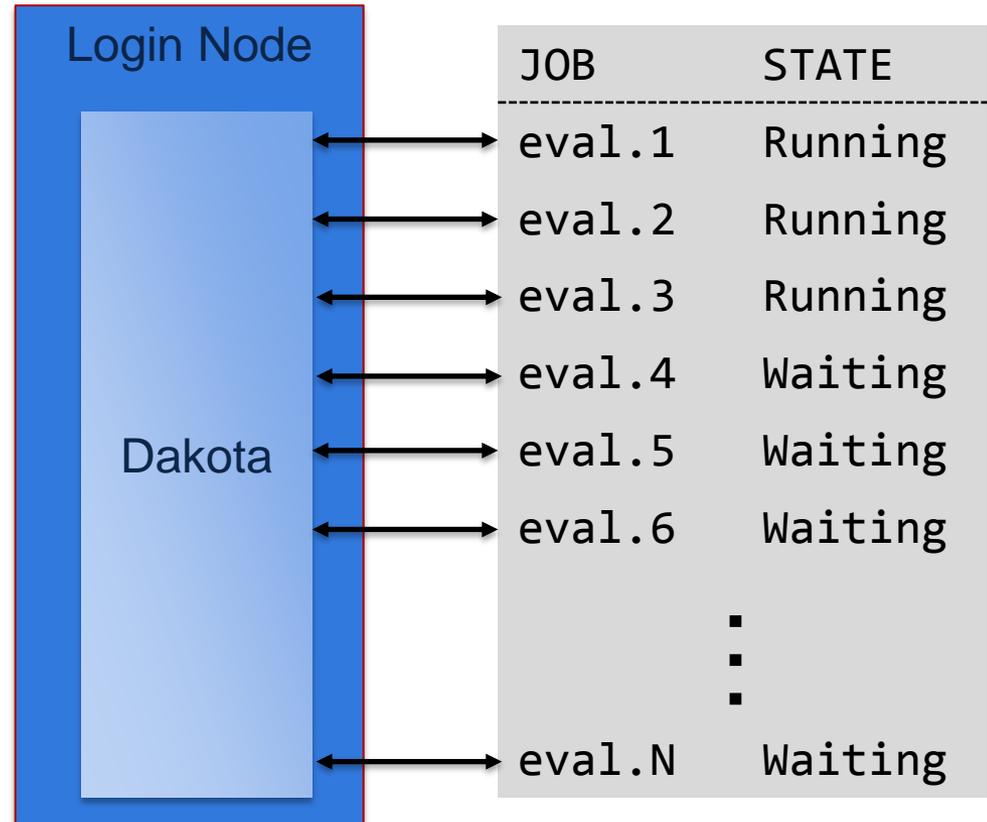
- Evaluation Submission
- Evaluation Tiling

# Approach 1: Evaluation Submission



## Evaluation Steps

1. Dakota invokes analysis driver as usual
2. Driver performs pre-processing
3. Driver submits a job to the queue and waits for it to finish
4. Job starts, runs the simulation, and finishes
5. Driver performs post-processing and exits
6. Dakota reads results file and continues



# Example Interface



## Analysis driver snippet

```
## Pre-processing done above (omitted)

sbatch eval.sbatch > sbatch.out

# Wait until the batch job finishes before
continuing.

jobid=$(tail -1 sbatch.out | egrep -o '[0-9]+')
while [ $(squeue -j $jobid | wc -l) -ne 0 ];
do
    sleep 300
done

## Post-processing done below (omitted)
```

## eval.sbatch

```
#!/bin/bash

#SBATCH --nodes=64
#SBATCH --time=08:00:00
#SBATCH --account=my_account
#SBATCH --job-name=eval.1

module load my_simulation

mpirun -np 1024 my_simulation
```



# Instead of waiting

When using ‘single-pass’ methods, Dakota can be run in two steps

- **Step 1: Job Creation**
  - Analysis driver set up to submit jobs then immediately exit, returning “dummy” values to Dakota
- **Step 2: Data Collection (after all jobs have finished)**
  - Analysis driver set up to post-process and return real result to Dakota

**Tip:** Dakota must generate the same parameters in both steps.  
For stochastic methods use the seed keyword.

# Recommended Dakota Input



```
interface
```

```
  analysis_driver "driver.sh"
```

```
  fork
```

```
  asynchronous
```

```
    evaluation_concurrency 20
```

Submit multiple jobs

```
  allow_existing_results
```

Prevent Dakota from erasing existing results

```
  work_directory "runs/run"
```

```
  directory_tag
```

```
  directory_save
```

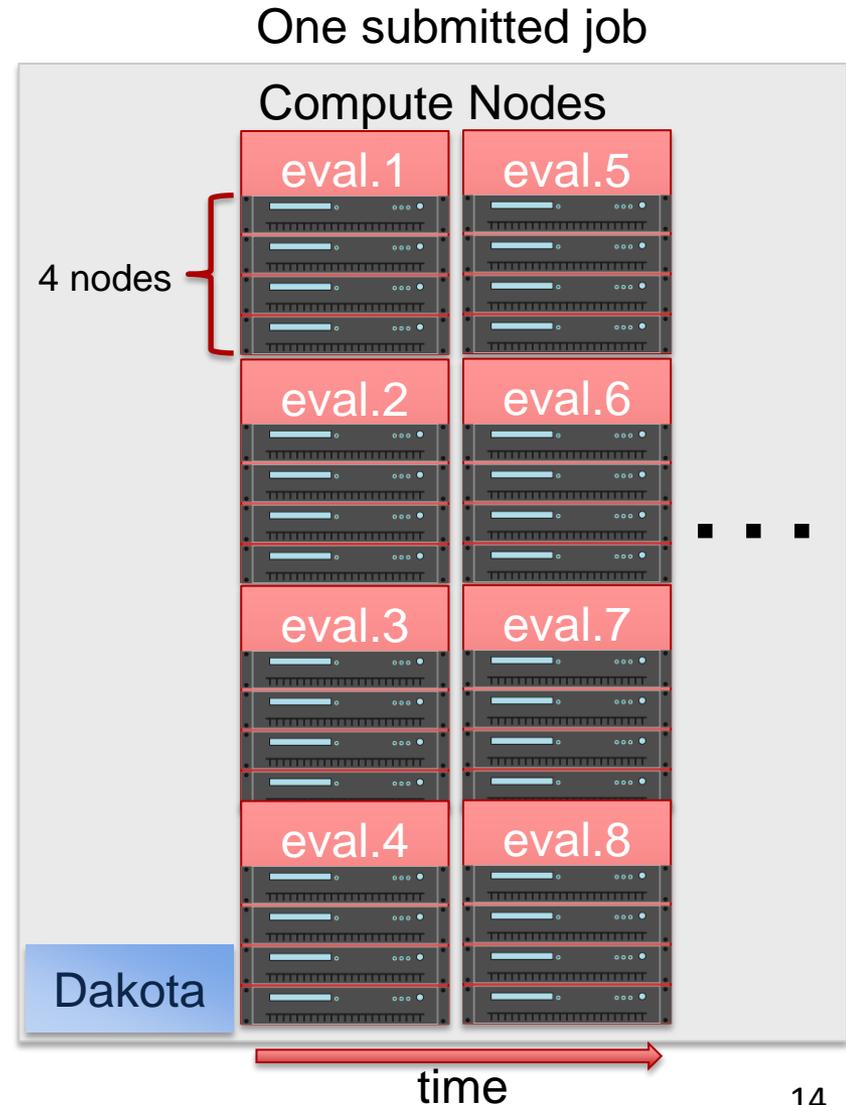
Keep simulation run files separate from one another and preserve run folders

# Approach 2: Evaluation Tiling



## Evaluation Steps

1. Dakota invokes analysis driver as usual
2. Driver performs pre-processing
3. Driver determines node placement (if necessary)
4. Driver launches parallel simulation
5. Driver performs post-processing and exits
6. Dakota reads results file and continues





# Node Placement Methods

## Automatic tiling

- just launch (srun, aprun)

## Relative node list or Machine files

- Compute list of relative nodes based on—
  - Number of nodes in allocation
  - Number of MPI tasks per node
  - Number of MPI tasks per simulation run
  - evaluation number (obtain from e.g. `file_tag`)
- Then launch simulation with relative node list option (`-host`) or machinefile option (`-machinefile`)
- Use `local_evaluation_scheduling static`
- Examples in
  - `examples/Case3_OpenMPI/`
  - `examples/Case3_MachinefileMgmt/`

# Example Analysis Driver



```
## Pre-processing done above (omitted)

APPLIC_PROCS=2
# Simple case: srun -n $APPLIC_PROCS my_simulation

num=$(echo $params | awk -F. '{print $NF}')
CONCURRENCY=4
PPN=16
applic_nodes=$(( (APPLIC_PROCS+PPN-1) / PPN ))
relative_node=$(( (num - 1) % CONCURRENCY * APPLIC_PROCS / PPN ))
node_list="+n${relative_node}"
for node_increment in $(seq 1 $((applic_nodes - 1)) ); do
  node_list="$node_list,+n$((relative_node + node_increment))"
done
mpirun -np $APPLIC_PROCS -host $node_list my_simulation

sleep 30
## Post-processing done below (omitted)
```

No. procs/simulation

No. concurrent evaluations

Procs per node

No. nodes required by simulation

0-based index of starting node

List of nodes where simulation will run

# Recommended Dakota Input



```
interface
```

```
analysis_driver "driver.sh"
```

```
fork
```

```
asynchronous
```

```
evaluation_concurrency 4
```

Run multiple concurrent evaluations

```
local_evaluation_scheduling static
```

Use static scheduling

```
file_tag
```

File tagging to extract evaluation number

```
work_directory "runs/run"
```

```
directory_tag directory_save
```

Keep simulation run files separate from one another and preserve run folders

# Tiling versus Submission



## Consider submission when..

- Memory or core count requirements are large
- Fork/system is disallowed on the compute nodes

## Consider tiling when..

- Memory or core count requirements are modest
- Using an adaptive method

# Examples and Documentation



- **Examples folder** (examples/parallelism)
- **User's Manual** (Chapter 17)
- **Note:** In these resources, running Dakota in parallel is referred to as “Case 1” parallelism, Evaluation Submission is “Case 4,” and Evaluation Tiling is “Case 3.” (Sorry.)