Orchestrating end-to-end reproducible NextGen and NextGen-adjacent workflows

Part 2: Orchestrating parallel model calibration

Workshop leads: Tristan Montoya, Darri Eythorsson, Jordan Read, James Halgren, Martyn Clark, and Raymond Spiteri

CIROH DevCon 2026

Expected outcomes

By the end of this session, you should be able to:

A. Reason about parallel computing and its applications in hydrology

- Understand the basic terms and concepts used in parallel computing
- Identify parallelizable work in hydrologic modeling tasks
- Interpret speedup, scaling behavior, efficiency, and common bottlenecks

B. Run parallel model execution and calibration tasks in SYMFLUENCE

- Calibrate a hydrologic model using a parallel optimization algorithm
- Run a distributed hydrologic model across multiple workers
- Connect notebook settings to the parallel work being executed

Running the example notebooks

1. Download the workshop materials (slides and notebooks) from the workshop repository (first pinned repository on `https://github.com/tristanmontoya`)

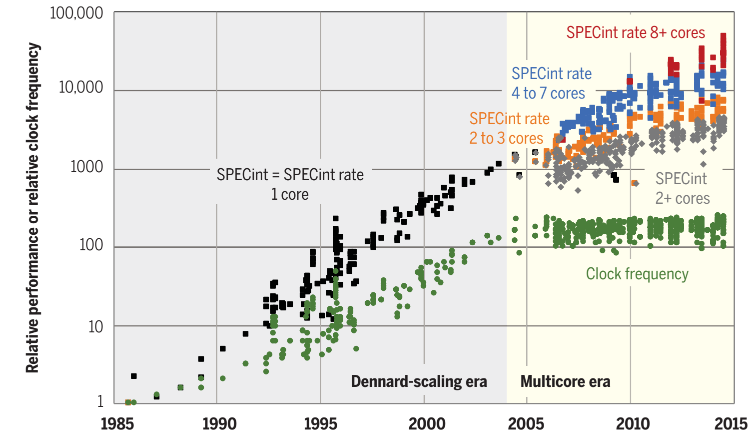
   `https://github.com/tristanmontoya/devcon-2026`

2. Follow the instructions in the `README.md` file (or the email you received yesterday) to set up your environment on 2i2c or locally on your machine
3. Choose your own adventure:
   - Run the pre-made notebooks `04a_logan_river_workshop_asyncdds.ipynb` and `04a_logan_river_workshop_distributed.ipynb` now to (hopefully) give them time to execute before we discuss them
   - Follow along by modifying `04a_logan_river_workshop.ipynb` as we go

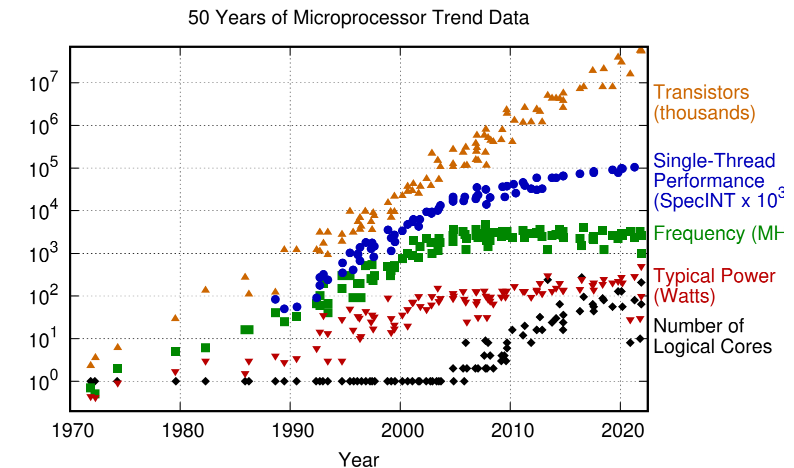
A. Introduction to parallel computing for hydrologists

What is parallel computing?

- **Parallel computing** is a type of computation in which many calculations or processes are carried out simultaneously on multiple processors or cores
- Closely related to **concurrency**, where multiple tasks can be in progress during the same period, but need not execute simultaneously
- **Increased parallelism, not clock speed**, has been responsible for most of the performance improvements in computing since the mid-2000s

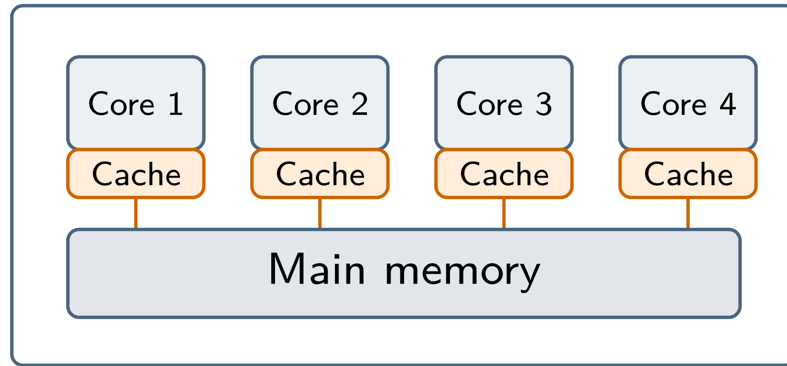


C. E. Leiserson et al. (2020). "There's Plenty of Room at the Top: What Will Drive Computer Performance After Moore's Law?" *Science* 368.6495

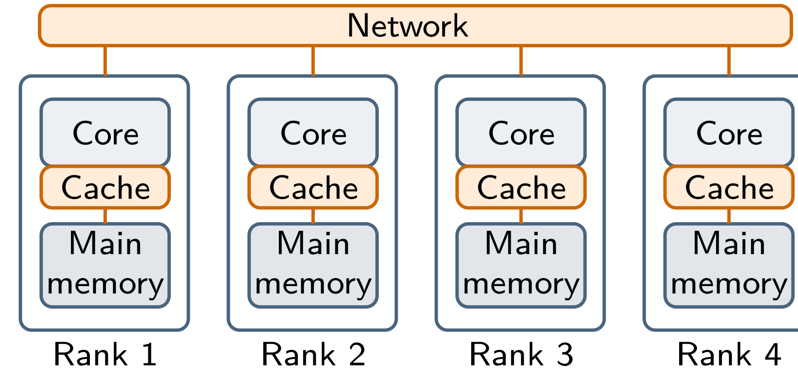


K. Rupp (2022). *Microprocessor Trend Data*. `https://github.com/karlrupp/microprocessor-trend-data`

Parallel computing paradigms



Shared memory

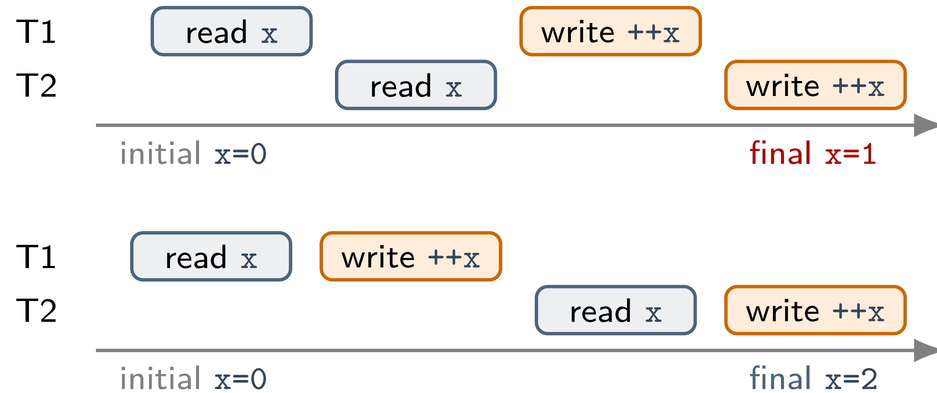


Distributed memory

- **Shared memory:** multiple cores executing threads on one machine, sharing a single memory space (e.g. multithreading, OpenMP)
- **Distributed memory:** multiple ranks, each with its own memory space, often on multiple nodes that communicate by network interconnect (e.g. MPI)
- Many parallel applications combine both paradigms (e.g. many cores per rank)
- Finer-grained parallelism is often exploited within each processor (e.g. vector registers) or using accelerators (e.g. GPUs)

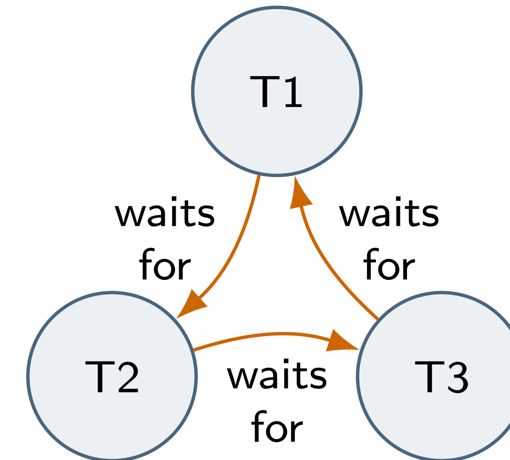
Hazards and correctness

Race condition



Unsynchronized accesses to shared data make the result depend on timing

Deadlock



Circular blocking dependencies leave threads stuck indefinitely

Thread safety: code retains correct behavior under use by multiple threads

Quiz: is this code thread safe? If not, how would you fix it?

```
import threading

N = 200_000
chunk_size = 50_000
total = 0

ranges = [(start, min(start + chunk_size, N + 1))
          for start in range(1, N + 1, chunk_size)]

def sum_chunk(start, stop):
    global total
    for i in range(start, stop):
        total += i

threads = [threading.Thread(target=sum_chunk, args=r)
           for r in ranges]

for t in threads: t.start()
for t in threads: t.join()

print(total)
```

Quiz: is this code thread safe? If not, how would you fix it?

```
import threading

N = 200_000
chunk_size = 50_000
total = 0

ranges = [(start, min(start + chunk_size, N + 1))
          for start in range(1, N + 1, chunk_size)]

def sum_chunk(start, stop):
    global total
    for i in range(start, stop):
        total += i

threads = [threading.Thread(target=sum_chunk, args=r)
           for r in ranges]

for t in threads: t.start()
for t in threads: t.join()

print(total)
```

Not thread safe due to **race condition** on shared variable `total`[1]

```
import threading

N = 200_000
chunk_size = 50_000
total = 0
lock = threading.Lock()

ranges = [(start, min(start + chunk_size, N + 1))
          for start in range(1, N + 1, chunk_size)]

def sum_chunk(start, stop):
    global total
    for i in range(start, stop):
        with lock:
            total += i

threads = [threading.Thread(target=sum_chunk, args=r)
           for r in ranges]

for t in threads: t.start()
for t in threads: t.join()

print(total)
```

Thread safe, but the lock serializes the update

[1] **Note:** this might still give the correct answer, but it is not guaranteed!

Quiz: is this code thread safe? If not, how would you fix it?

```
import threading

N = 200_000
chunk_size = 50_000
total = 0

ranges = [(start, min(start + chunk_size, N + 1))
          for start in range(1, N + 1, chunk_size)]

def sum_chunk(start, stop):
    global total
    for i in range(start, stop):
        total += i

threads = [threading.Thread(target=sum_chunk, args=r)
           for r in ranges]

for t in threads: t.start()
for t in threads: t.join()

print(total)
```

Not thread safe due to **race condition** on shared variable `total`[1]

```
import threading

N = 200_000
chunk_size = 50_000

ranges = [(start, min(start + chunk_size, N + 1))
          for start in range(1, N + 1, chunk_size)]

subtotals = [0] * len(ranges)

def sum_chunk(k, start, stop):
    for i in range(start, stop):
        subtotals[k] += i

threads = [threading.Thread(target=sum_chunk, args=(k, *r))
           for k, r in enumerate(ranges)]

for t in threads: t.start()
for t in threads: t.join()

total = sum(subtotals)

print(total)
```

Thread safe and avoids serialization by using separate subtotals

[1] **Note:** this might still give the correct answer, but it is not guaranteed!

Performance and speedup

- Several factors limit the performance gains of parallel computing:
    1. **Limited parallelism:** not all parts of a computation can execute simultaneously
    2. **Communication overhead:** processors must wait for data exchange
    3. **Synchronization and blocking:** processors may need to wait for each other to coordinate
    4. **Load imbalance:** some processors finish early while others remain busy (straggle)
- Suppose that a computation of size N takes time $T_1(N)$ on one processor and time $T_p(N)$ on p processors. The **speedup** is

$$S_p(N) = \frac{T_1(N)}{T_p(N)}$$

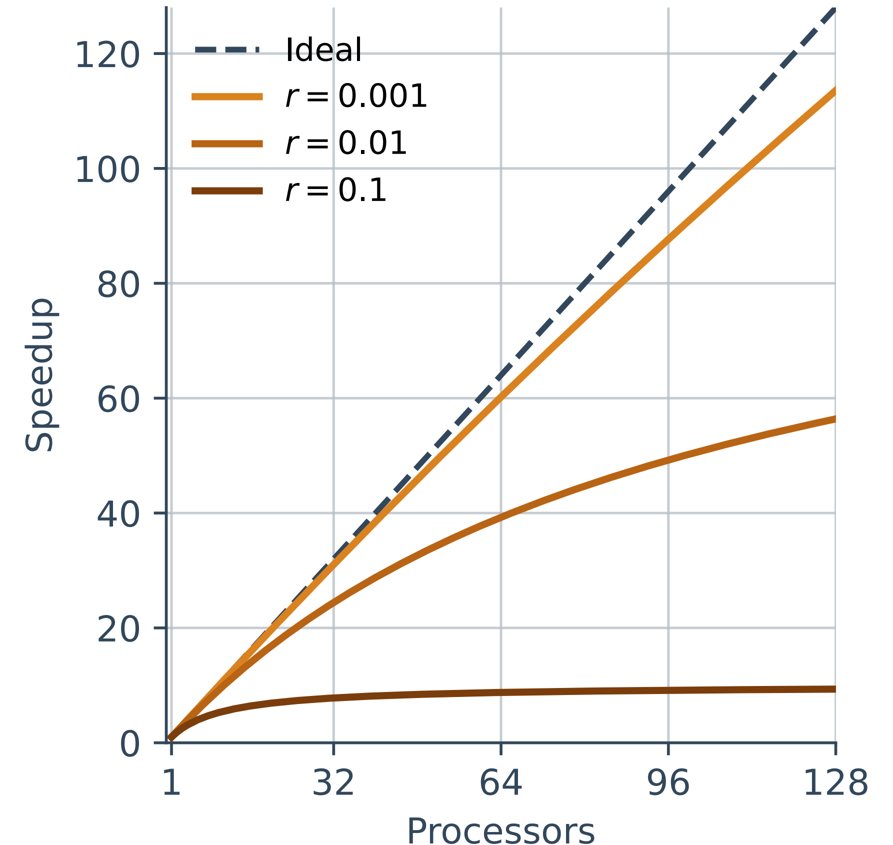
- Ideally, we would have **perfect speedup** ($S_p(N) = p$), but in practice this is rare

A fundamental limitation

- Suppose a program can mostly run in parallel, but a fraction r of it must remain sequential. Then the sequential part limits the overall speedup.
- This principle is often called **Amdahl's law**, and provides a bound on the speedup that can be achieved when holding the problem size fixed:

$$S_p(N) \leq \frac{1}{r + (1-r)/p}$$

- No matter how many processors we use, the speedup is at most $1/r$ (e.g. $10\times$ for $r = 0.1$)



Maximum speedup from Amdahl's law

Strong scaling, weak scaling, and efficiency

- **Strong scaling:** how does the time to solve a fixed-size problem change as we increase the number of processors? Efficiency is measured by how close the speedup is to perfect:

$$E_p(N) = \frac{S_p(N)}{p}$$

- **Weak scaling:** how does the time to solve a problem change as we increase the number of processors while keeping the workload per processor constant at n such that $N = pn$? Efficiency is measured by how close the time remains to constant:

$$E_p(N) = \frac{T_1(N/p)}{T_p(N)}$$

- **Embarrassingly parallel:** a problem whose sub-tasks can be executed independently, with negligible communication between processors

Quiz: parallel efficiency and performance considerations

1. Are measured efficiencies typically higher in strong or weak scaling tests? Why?

2. What are some examples of embarrassingly parallel computations in hydrology?

3. What important performance questions cannot be answered by parallel efficiency alone?

Quiz: parallel efficiency and performance considerations

1. Are measured efficiencies typically higher in strong or weak scaling tests? Why?

   Weak scaling efficiencies are usually higher because work per processor stays fixed. In strong scaling, communication, synchronization, and serial work take up a larger share of runtime as the number of processors increases.

2. What are some examples of embarrassingly parallel computations in hydrology?

3. What important performance questions cannot be answered by parallel efficiency alone?

Quiz: parallel efficiency and performance considerations

1. Are measured efficiencies typically higher in strong or weak scaling tests? Why?

   Weak scaling efficiencies are usually higher because work per processor stays fixed. In strong scaling, communication, synchronization, and serial work take up a larger share of runtime as the number of processors increases.

2. What are some examples of embarrassingly parallel computations in hydrology?

   Many hydrologic computations such as parameter sweeps, calibration procedures, ensemble runs, independent watershed or catchment simulations, and processing of gridded hydrologic data can often be embarrassingly parallel.

3. What important performance questions cannot be answered by parallel efficiency alone?

Quiz: parallel efficiency and performance considerations

1. Are measured efficiencies typically higher in strong or weak scaling tests? Why?

   Weak scaling efficiencies are usually higher because work per processor stays fixed. In strong scaling, communication, synchronization, and serial work take up a larger share of runtime as the number of processors increases.

2. What are some examples of embarrassingly parallel computations in hydrology?

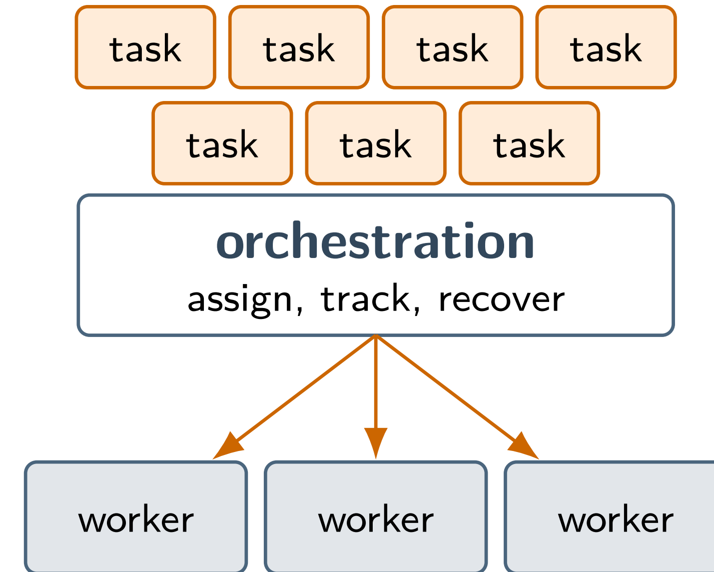
   Many hydrologic computations such as parameter sweeps, calibration procedures, ensemble runs, independent watershed or catchment simulations, and processing of gridded hydrologic data can often be embarrassingly parallel.

3. What important performance questions cannot be answered by parallel efficiency alone?

   A parallel program can have high efficiency despite (or because of) a poor underlying algorithm, for example, one that does unnecessary work or imposes excessive memory requirements. A scalable algorithm may still be slow if its constituent tasks have poor serial performance.

Orchestration

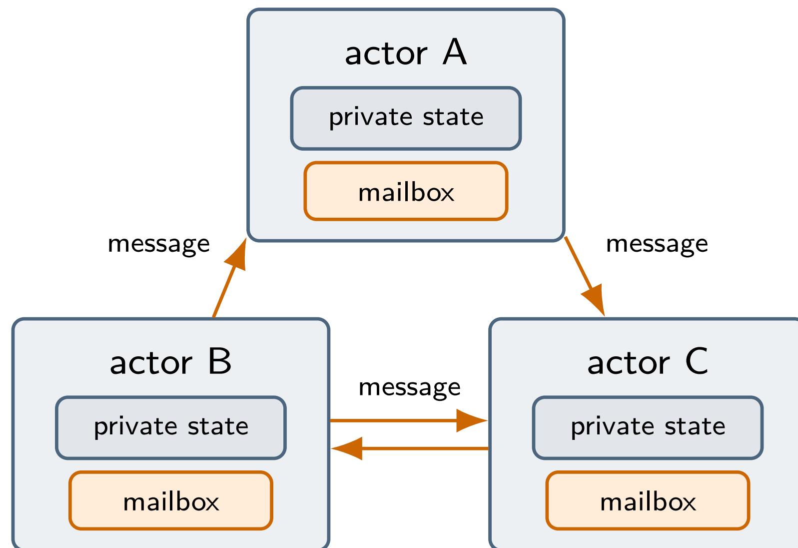
- Involves assigning tasks to workers (e.g. threads or ranks), tracking progress, and handling failures
- Especially important when there are many more tasks than workers
- Objective is to keep workers busy and minimize idle time, while ensuring correct results and isolating failures



Example: Continental-scale hydrologic modeling with SUMMA – Many independent simulations with differing runtimes must be scheduled on a smaller set of cores; requires orchestration to keep cores busy, handle stragglers/failures

The actor model of concurrent computing

- Decompose the program into independent **actors**[2] that run concurrently
- Actors do not share state; they communicate by asynchronous messages
- While handling a message, an actor can **create** new actors, **send** new messages, or **become** a new type of actor with different behavior

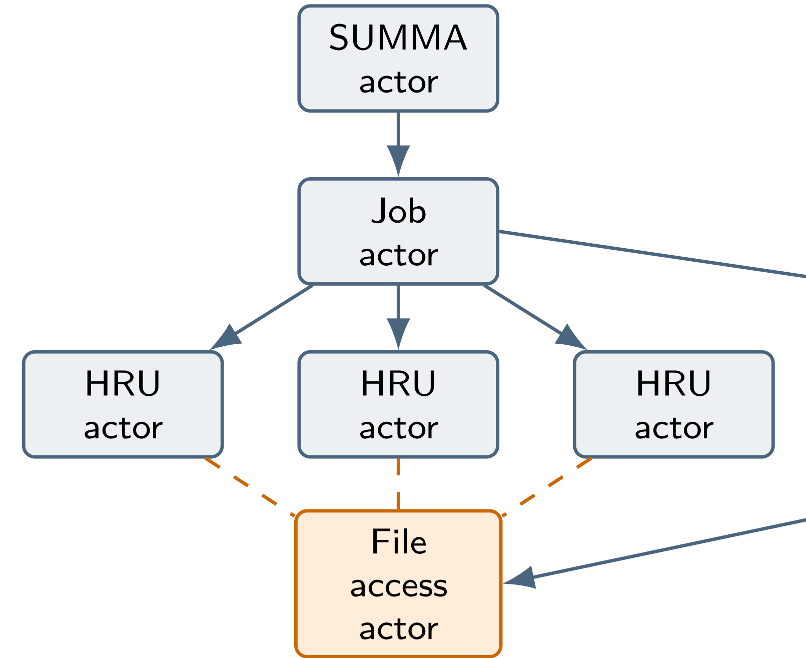


- High-level abstraction for orchestration that hides hardware-dependent details
- Underlying runtime handles scheduling at the worker level (separation of concerns)
- Flexible, decentralized design can improve scalability and fault tolerance
- Encapsulated state can reduce risk of race conditions

[2] C. Hewitt et al. (1973). "A Universal Modular ACTOR Formalism for Artificial Intelligence." *Proceedings of the Third International Joint Conference on Artificial Intelligence*, pp. 235–245.

SUMMA-Actors

- **SUMMA actor:** represents the large-domain model run
- **Job actor:** coordinates a submitted batch of hydrologic response unit (HRU) tasks
- **HRU actor:** runs one HRU task
- **File access actor:** coordinates input/output access



- Submit all HRUs at once instead of manually tuning batches for each cluster
- Dynamic scheduling reduces idle cores when HRU runtimes are imbalanced
- Wall-clock time for 500,000+ HRUs improved by 10–26% compared with job arrays and GNU Parallel; no need for user intervention to resubmit failed jobs

B. Parallel hydrologic model calibration in SYMFLUENCE

What do we mean by parallel model calibration?

- Recall the calibration problem we are trying to solve:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta} \in [0,1]^N}{\arg\max}\ \mathrm{KGE}(\mathbf{q}_s(\boldsymbol{\theta}), \mathbf{q}_o)$$

- $\boldsymbol{\theta}$ is the normalized calibration vector, mapped linearly or logarithmically from physical parameter ranges
- $\mathbf{q}_s(\boldsymbol{\theta})$ and $\mathbf{q}_o$ are simulated and observed streamflow over the calibration period
- KGE is the standard Kling–Gupta efficiency, defined as

$$\mathrm{KGE} = 1 - \sqrt{(r-1)^2 + (\alpha-1)^2 + (\beta-1)^2},$$

where r is the correlation, α is the variability ratio, and β is the mean bias ratio

We can exploit parallelism **across trials in the optimization algorithm** or **across spatially distributed features** in the model used to compute $\mathbf{q}_s(\boldsymbol{\theta})$

Overview of dynamically dimensioned search (DDS)

1. Initialize one incumbent candidate
   - Choose $\boldsymbol{\theta}^0 \in [0,1]^N$ from an initial guess or by random sampling
   - Evaluate $\mathsf{KGE}(\mathbf{q}_s(\boldsymbol{\theta}^0), \mathbf{q}_o)$
   - Store $\boldsymbol{\theta}^0$ as the current best solution
2. While the stopping criteria are not met, evaluate one candidate at a time
   - Select a subset $S \subseteq \{1, \ldots, N\}$ of dimensions to perturb, each with probability

     $$p = \max\left(1 - \frac{\log(i)}{\log(M)}, \frac{1}{N}\right),$$

     where i is the current trial number and M is the target evaluation budget
   - Perturb selected dimensions S according to a normal distribution with standard deviation `DDS_R`, then reflect and clip into $[0,1]$ to obtain the new candidate $\boldsymbol{\theta}^i$ to evaluate
   - Compute $\mathsf{KGE}(\mathbf{q}_s(\boldsymbol{\theta}^i), \mathbf{q}_o)$ for the new trial
   - If the trial improves KGE, replace the incumbent with $\boldsymbol{\theta}^i$
3. Return the best incumbent candidate as $\boldsymbol{\theta}^*$

Calibration with parallel trials (ASYNC_DDS)

1. Broker builds the initial pool of candidates; workers evaluate them in parallel
    - Broker draws ASYNC_DDS_POOL_SIZE initial candidate parameter vectors $\boldsymbol{\theta}^k \in [0,1]^N$
    - Parallel workers evaluate $\text{KGE}(\mathbf{q}_s(\boldsymbol{\theta}^k), \mathbf{q}_o)$ for each candidate
    - Broker sorts valid candidates by $\text{KGE}(\mathbf{q}_s(\boldsymbol{\theta}^k), \mathbf{q}_o)$ and stores them as the initial pool
2. While the stopping criteria are not met, generate and dispatch batches of candidates
    - For each trial index $j \in \{1, \ldots, \texttt{ASYNC_DDS_BATCH_SIZE}\}$
        - Pick up to three candidates from the pool and use the best-scoring sample $\boldsymbol{\theta}_j^*$ as the parent
        - Select a subset $S \subseteq \{1, \ldots, N\}$ of dimensions to perturb, each with probability

$$p = \max\left(1 - \frac{\log(m+j)}{\log(M)}, \frac{1}{N}\right),$$

          where m is the completed evaluation count before this batch and M is the target evaluation budget
        - Perturb selected dimensions S according to a normal distribution with standard deviation DDS_R, then reflect and clip into $[0,1]$ to obtain the new candidate $\boldsymbol{\theta}_j$ to add to the batch
    - Parallel workers evaluate $\text{KGE}(\mathbf{q}_s(\boldsymbol{\theta}_j), \mathbf{q}_o)$ for each candidate and return the results to the broker
    - Broker merges the returned scores into the evaluated pool, sorts by score, and keeps the best ASYNC_DDS_POOL_SIZE candidates
3. Return the best-scoring candidate in the final pool as $\boldsymbol{\theta}^*$

Notebook changes

- We will walk through two stages of notebook changes to illustrate how to configure parallel calibration and semi-distributed modelling in SYMFLUENCE
- A more complex process-based model (**SUMMA**) will be used to expose more parallel work (although you can still use HBV)

1. Parallel calibration trials

Reference: `04a_logan_river_workshop_asyncdds.ipynb`

- Replace HBV with lumped SUMMA
- Set SUMMA-specific parameters
- Set explicit parameter bounds
- Run `ASYNC_DDS` using multiple processes

2. Parallel semi-distributed modelling

Reference: `04a_logan_river_workshop_distributed.ipynb`

- Switch to the semi-distributed Logan River domain
- Enable routing with mizuRoute
- Enable parallel SUMMA execution
- Switch back to standard `DDS`

Quiz: scaling in parallel model calibration

1. Which model is more likely to demonstrate better strong scaling in this workshop: HBV or SUMMA? Why?

2. A serial semi-distributed SUMMA run spends 4 minutes in SUMMA and 1 minute in routing. With 16 workers, assume SUMMA scales perfectly across GRUs but routing stays serial. What speedup and efficiency can we expect? During calibration, should we add workers to each run, or should we run more SUMMA trials concurrently? Why?

Quiz: scaling in parallel model calibration

1. Which model is more likely to demonstrate better strong scaling in this workshop: HBV or SUMMA? Why?

   SUMMA, because each model evaluation does more work, especially in the semi-distributed setup. HBV runs are cheaper, so scheduling, startup, communication, and I/O overheads take up a larger share of runtime.

2. A serial semi-distributed SUMMA run spends 4 minutes in SUMMA and 1 minute in routing. With 16 workers, assume SUMMA scales perfectly across GRUs but routing stays serial. What speedup and efficiency can we expect? During calibration, should we add workers to each run, or should we run more SUMMA trials concurrently? Why?

Quiz: scaling in parallel model calibration

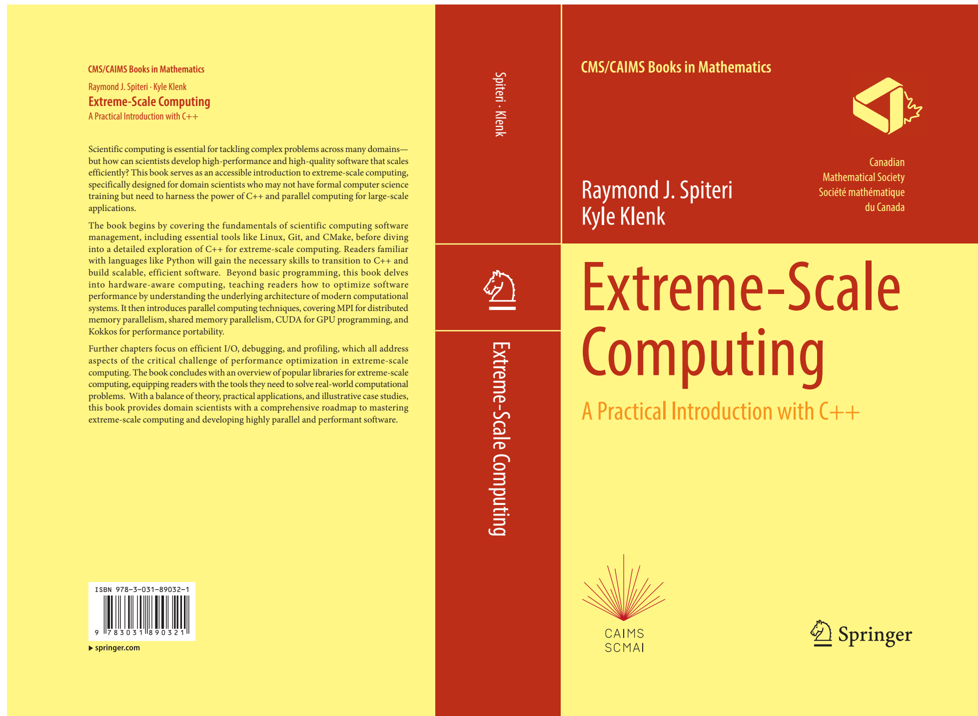
1. Which model is more likely to demonstrate better strong scaling in this workshop: HBV or SUMMA? Why?

   SUMMA, because each model evaluation does more work, especially in the semi-distributed setup. HBV runs are cheaper, so scheduling, startup, communication, and I/O overheads take up a larger share of runtime.

2. A serial semi-distributed SUMMA run spends 4 minutes in SUMMA and 1 minute in routing. With 16 workers, assume SUMMA scales perfectly across GRUs but routing stays serial. What speedup and efficiency can we expect? During calibration, should we add workers to each run, or should we run more SUMMA trials concurrently? Why?

   The 16-worker runtime is $4/16 + 1 = 1.25$ minutes, so speedup is $5/1.25 = 4\times$ and efficiency is $4/16 = 25\%$. Since the limiting speedup is only $5\times$, extra workers per run have little value; use them for more concurrent SUMMA trials unless I/O or scheduling becomes limiting.

References



High-performance computing

R. J. Spiteri and K. Klenk (2025). *Extreme-Scale Computing: A Practical Introduction with C++*. Springer

SUMMA

M. P. Clark et al. (2015a). "A Unified Approach for Process-Based Hydrologic Modeling: Part 1. Modeling Concept." *Water Resources Research*

M. P. Clark et al. (2015b). "A Unified Approach for Process-Based Hydrologic Modeling: Part 2. Model Implementation and Case Studies." *Water Resources Research*

M. P. Clark et al. (2021). "The Numerical Implementation of Land Models: Problem Formulation and Laugh Tests." *Journal of Hydrometeorology*

SUMMA-Actors

K. Klenk and R. J. Spiteri (2024). "Improving Resource Utilization and Fault Tolerance in Large Simulations via Actors." *Cluster Computing* 27, pp. 6323–6340