WRF Computation

Understanding how to get the most computational performance

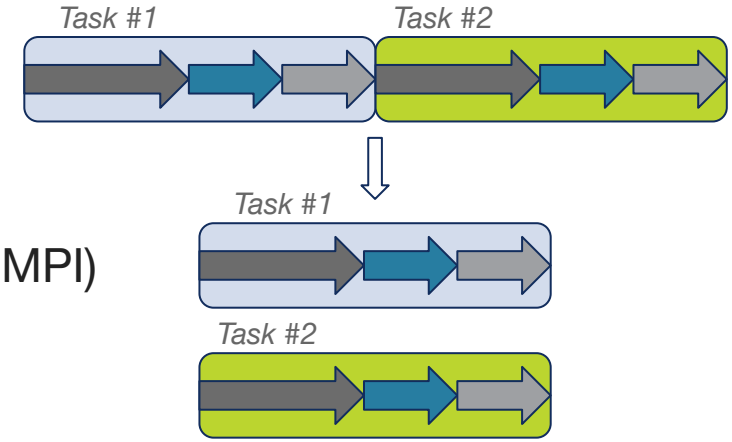
Anthony Islas,
WRF Tutorial

Agenda

Fundamentals
- Latency
- Memory Hierarchy

Parallel Processing
- Shared Memory vs Distributed (OpenMP vs MPI)

WRF Parallelism
- Domain Decomposition
- Halo Exchanges
- Scaling

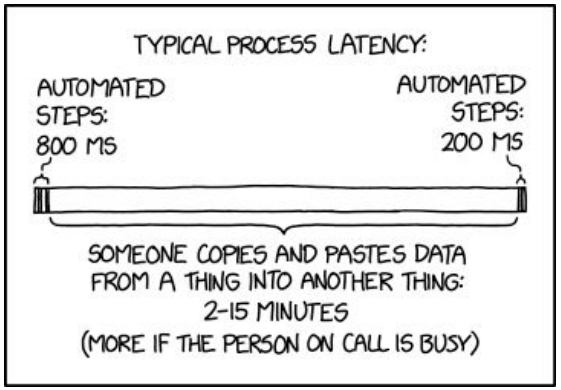


Latency

With an infinitely fast computer, no timing optimization or parallel execution would ever be necessary (memory available permitting)

We are limited to the speed at which physical interactions of electricity propagate within silicon/copper/etc.

- This delay continuously adds up, compounding with distance and complexity (# of interactions)
- Ultimately when increasing computational speed, we are fighting latency at many levels



xkcd #2565 Latency

For a single task, to go as fast as possible we should:

- Try to keep actions as close to minimum delay as possible
- When required to slow down to longer delays, limit the time in this state

Building an Analogy for Latency

Assume our computation speed is as fast as a human can reasonably move. Our goal is making a sandwich. Seems generally easy, right?

- The fastest would be to have all the ingredients ready on the table
- The slowest would be going to the store to get the ingredients one at a time

As we increase tasking the workload on a single processor can be immense. If we have access to more resources we can increase number of workers

- Each worker would now have less overall work to do vs the single processor

Using the analogy, this could be like adding more kitchen workspaces, pantry space, etc.

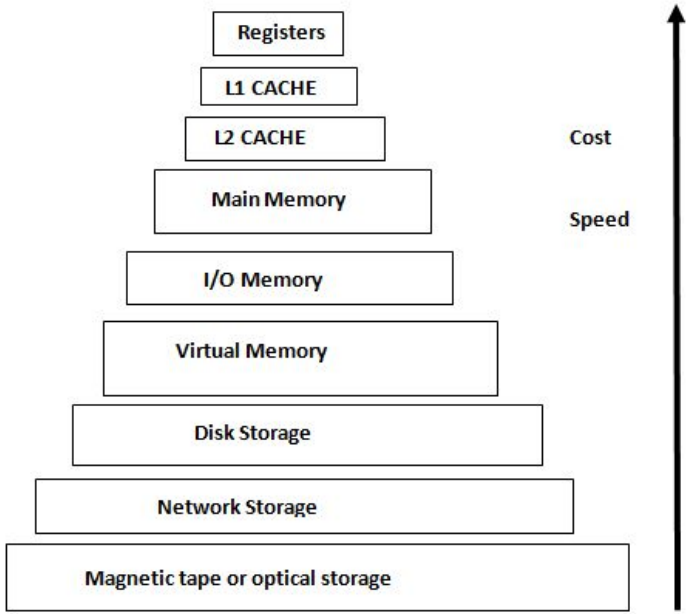
Processors can only go so fast, and certain actions always take longer

But with good division of labor and resources things can be accelerated

Memory Hierarchy

Computer Memory Hierarchy - building an intuition without numbers

- How fast is fast?
- How slow is slow?
- How much time is long/short term?



Using our analogy, the speed and longevity of items could be represented as:

- CPU registers : Ingredients in your hands
- Cache : Ingredients on the counter
- RAM : Ingredients in the pantry/refrigerator
- Flash/SSD : Ingredients at the store on display
- Hard drive : Ingredients at local warehouse to stock
- Backup archive : Wait for ingredients to grow on farm

It is in our best interest to be efficient about when and how we access our “ingredients” for computation to minimize large-delay bottlenecks in our computation of “making a sandwich”

Shared vs Distributed Memory

Parallelization of workloads generally have an added layer of software coordination such as :

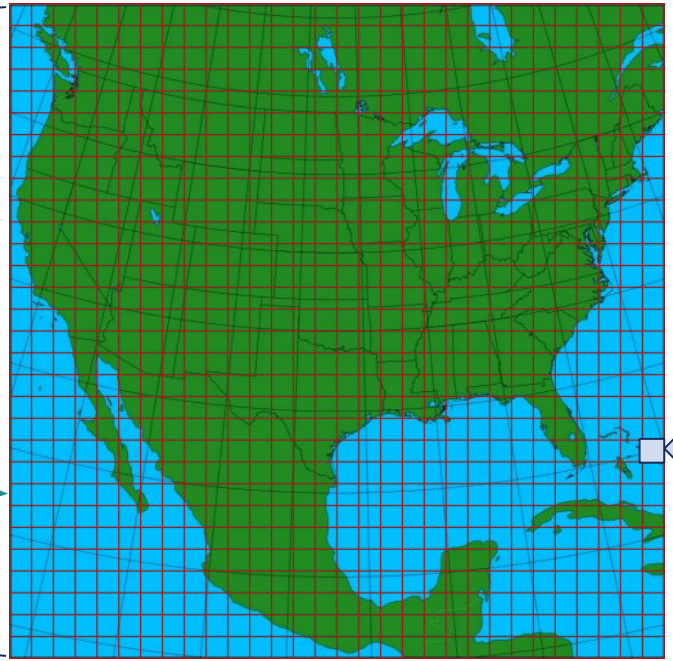
- OpenMP
  Shared memory (threads) operate within the same memory space of the task at hand
  Re: our analogy this could be equivalent to working on the same kitchen table
  - CPU memory space permitting, this can generally be faster
  - Not many slots available though and doesn't scale at large (e.g. one giant table)

- MPI
  Distributed memory (processes) operate within separate memory spaces and must use external coordination between tasks
  Re: our analogy this could equivalent to many separate kitchen tables, requiring coordination to happen away from the table
  - Operation outside the same memory space *could* take longer, isolation of memory usage could streamline resource access
  - Scales well at larger # of tasks every task isolated and is primarily limited by coordination layer (aside from actual computation) at this scale

2D Domain Decomposition

To effectively parallelize work in WRF, we need to break down the work into smaller pieces, ideally all similar to more easily distribute and divide the problem

```
&domains
...
 e_we    = 1500
 e_sn    = 1500
 dx      = 3000
 dy      = 3000
...
```



Choosing dx dy is often determined by resolution of feature size

Number of **cells** is often determined by region of interest, but we will see that this can sometimes be adjusted for better performance on nested domains

For a single non-nested domain:

- dx dy composes the area of a grid **cell**
- e_we e_sn determines the **total number** of cells

The smallest unit of computation is a single grid **cell**, each of which can be *independently parallelized* (until boundary must be updated)

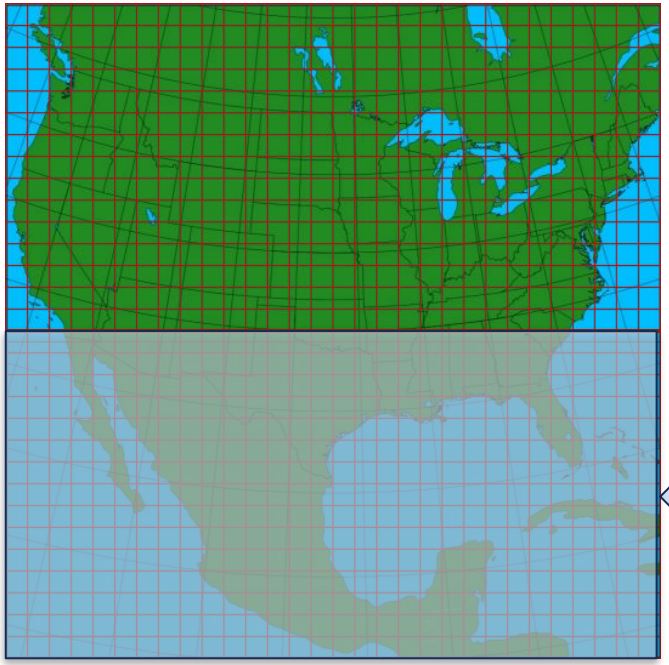
2D Domain Decomposition

OpenMP vs MPI - How to divide up the domain?

With (2) OpenMP Threads
We split the domain
in half to make two **tiles** -
two areas within the
same shared memory
allowing seamless communication
of boundary exchanges

This means the **tiles** don't
need to coordinate boundary
information transfer
between timesteps

However we still need to synchronize!

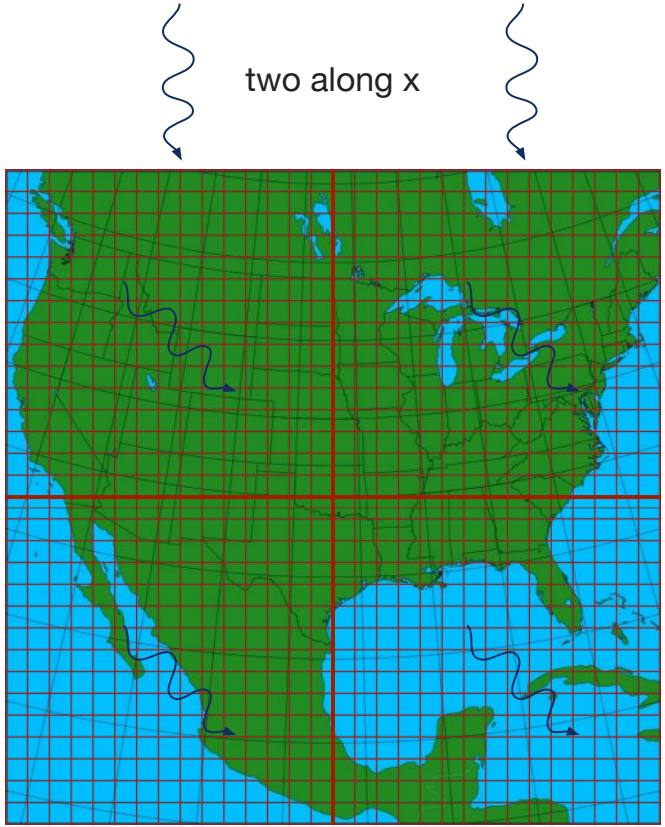


Legend / Terminology

Process (MPI) -

Thread (OpenMP) -

2D Domain Decomposition



We can continue adding threads to further decompose our domain.

This will make each thread's work smaller and thus overall execution time faster

However, eventually…

Legend / Terminology

Process (MPI) -

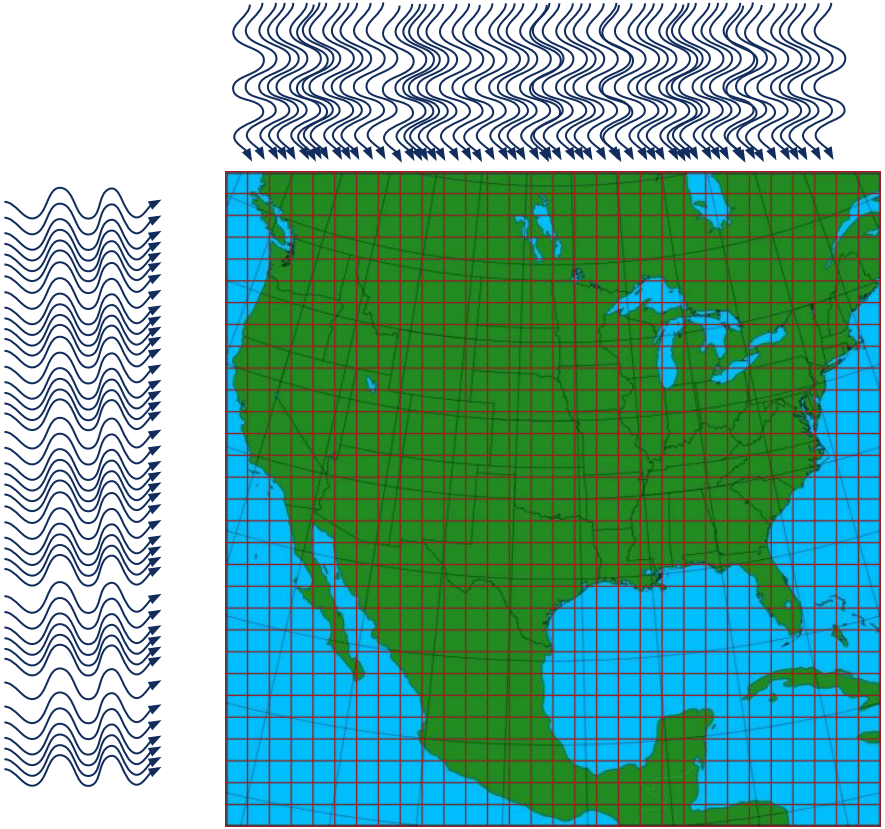
Thread (OpenMP) -

Note that the default is to use the **two closest factors** with the larger one in the j/ns/y direction. We can control this via :
tile_sz_x/y for OpenMP
nproc_x/y for MPI
But for most applications the default will work okay

2D Domain Decomposition

If our processor only has room for N threads performance can drop due to oversubscribed resources

Eventually we are at the limit of what an individual processor can handle, i.e. we need more resources than what a single processor can offer



Legend / Terminology

Process (MPI) -

Thread (OpenMP) -

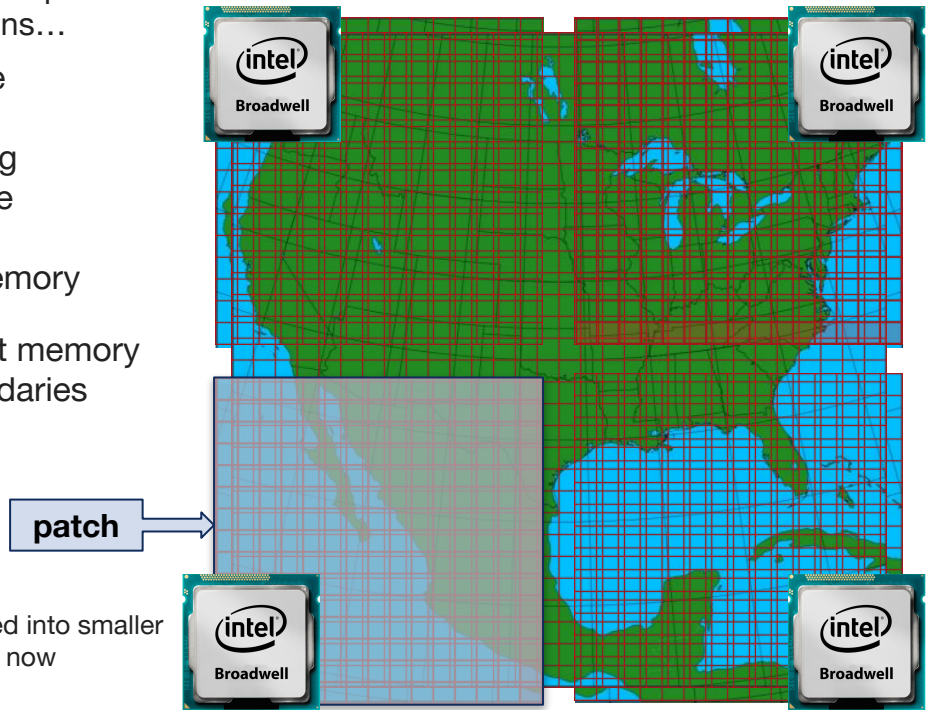
Recall our analogy : At a certain point, tasking more people to do a job without increasing *appropriate* resources leads to congestion and resource contention

2D Domain Decomposition

For better performance at larger scales we might want to allow our divisions of our domain to operate in separate memory locations…

We can instead allocate more resources per division by instead using MPI processes to create **patches** - areas of separate distributed memory

These reside in different memory spaces - now our boundaries cannot communicate with each other immediately…



Legend / Terminology

Process (MPI) -

Thread (OpenMP) -

Take for instance this **patch**

Its boundary resides in other **patches** outside its memory space

These *can* be further separated into smaller **tiles** but we will avoid that for now

Halo Exchanges

Halo exchanges are a necessary incurred latency cost to coordinating **patches**

This is a critical component of how well WRF scales

Recall our analogy :
Even if we had more processes, if we had each request handled separately, we'd pay the latency cost for each.

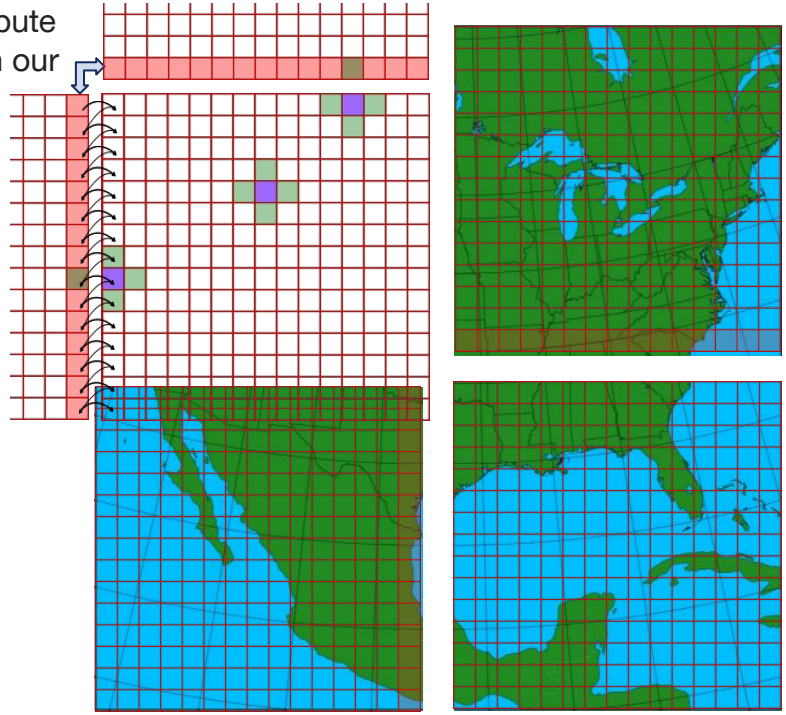
It would be like sending 15 people to the store to all get separate items in the same aisle for the same dish!

We need these **cells** to compute tendencies in our **patch**

Let's look at an example of computing some regions

We might assume we can individually transfer all **cells** independently at the same time…

But in reality if we only have one process, it would have to go back-and-forth between computations paying the cost each time



Halo Exchanges

Halo exchanges are a necessary incurred latency cost to coordinating **patches**

This is a critical component of how well WRF scales

So now we know how to effectively do **halo exchanges**

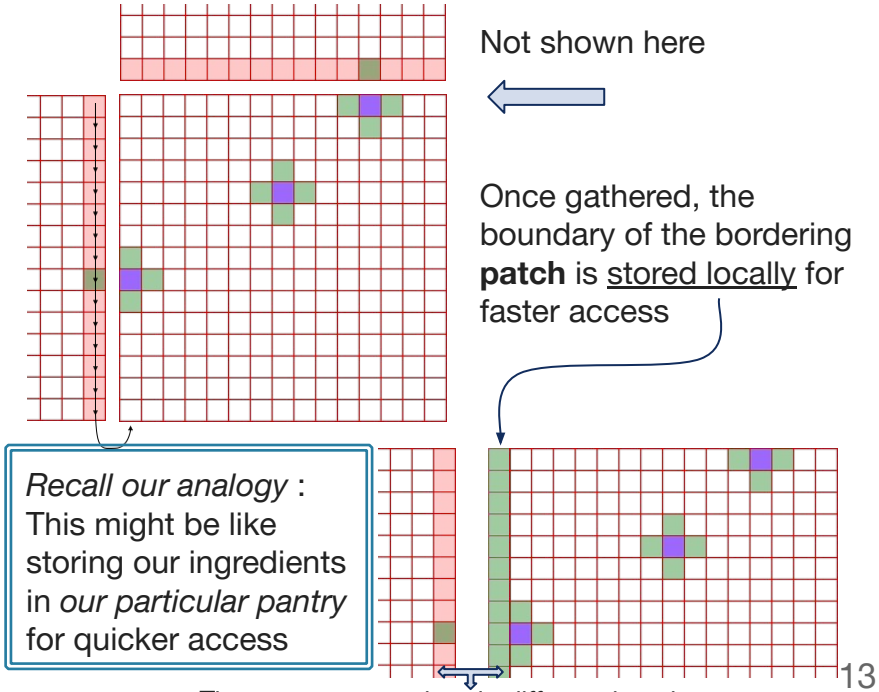
Why do we need to talk about scaling? Can't we just add more processes to go proportionally faster?

Let's instead optimize our "route" to gather values

Instead we can gather the **cells** all at once

This allows us to traverse the other **patch** in *its local memory space*, reducing latency

Now for any calculation along our **patch** border we no longer need to go out to a different memory space

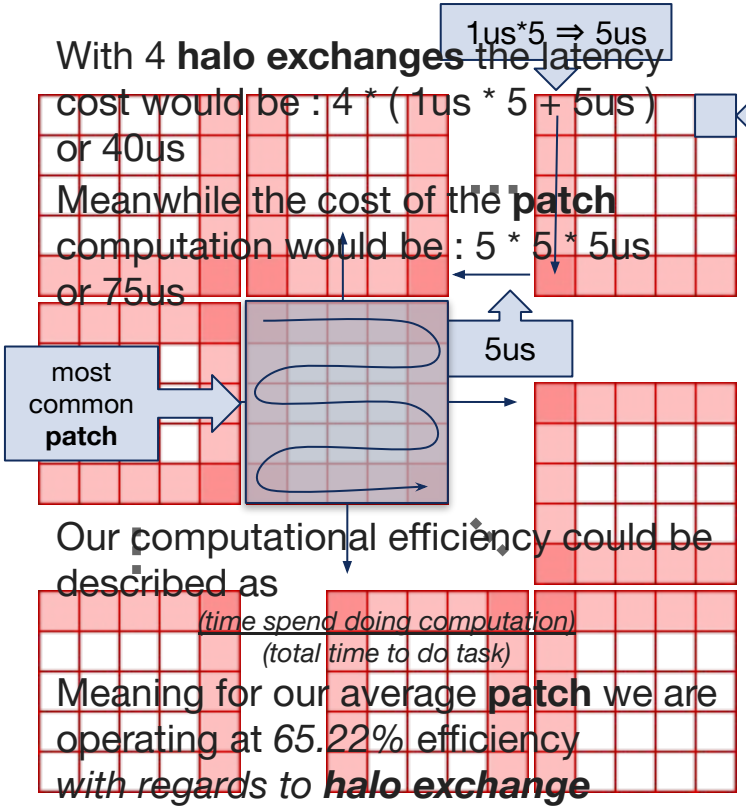


Limitations of Halo Exchanges

If we continue to subdivide our domain into **patches** with more MPI tasks, we can start to see how the proportion of our boundary coordination begins to affect our performance gains

Thought experiment:
Using the shown number of **cells** in the diagrams, let's say computation of
- one **cell** takes 5 us
- aggregation of a **cell** for a **halo exchange** is 1 us
- the **halo exchange** itself takes 5 us

**These are arbitrary generalized values*



With 4 **halo exchanges** the latency cost would be : 4 * (1us * 5 + 5us) or 40us

Meanwhile the cost of the **patch** computation would be : 5 * 5 * 5us or 75us

Our computational efficiency could be described as

(time spend doing computation)
(total time to do task)

Meaning for our average **patch** we are operating at *65.22%* efficiency *with regards to* ***halo exchange***

In our 1 process sit[illegible], while slow, all time is spent computing with no overhead on parallel coordination

Using an example **patch** of 15x15 **cells** our efficiency would be :
- 4 **halo exchanges**
  - each gathers 15 **cells**
- 15x15 **cells** computed

Yielding (225us)/(305us) or *73.77%* efficiency

Slower, but more efficient..

Limitations of Halo Exchanges

Smaller **patches** have decreased model runtime at the cost of lowered computational efficiency in part due to **halo exchange** and disproportionate shrinkage of area to perimeter, whereby the perimeter latency does not contribute to computational progress. Conversely, larger **patches** decrease speed but increase efficiency.

Halo exchanges are *NOT* the only form of latency and sole driving factor in reduction of efficiency, however they are a critical component and the concepts shown here further illustrate how other latencies may affect performance.

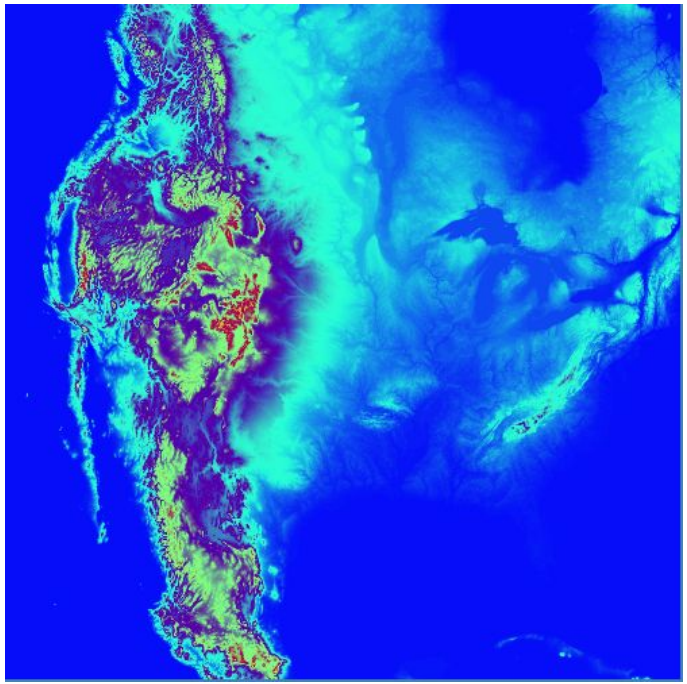
How does this all relate to actual runtimes? We know that if we add 2x processes, our speed up will likely not be 2x, but what will it actually be? How do we balance how many cores to allocate to get the desired speed increase?

Scaling - Model Configuration

WRF v4.5.2 1500x1500x50, 3km, CONUS Suite

- Domain
  - 1500 w/e x 1500 s/n
  - 50 vertical levels
  - 3km in dx and dy for **cells**
- Timesteps
  - 18s dt
  - 180s dt for rad steps
  - 12 minute simulation
  - 10hr spin-up then restart
- Options
  - No cumulus
  - Hybrid vertical activated
  - Moist theta
- No I/O done during timings

Run on Derecho 128-core nodes



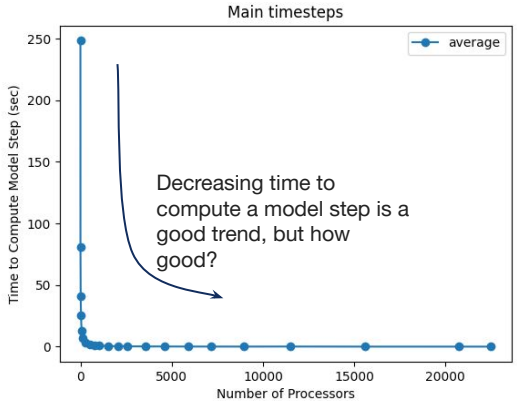
metgrid output of domain

Scaling

Achieving perfect 1-to-1 performance increase with added parallelism would be the theoretical maximum, however recall that latency and parallel coordination overhead thwart our numbers!

The closest to perfect efficiency we can be in terms of no additional latencies is a single process doing everything slowly. We can use this as a baseline to create two types of plots to explore how WRF actually scales with respect to real domain decomposition :

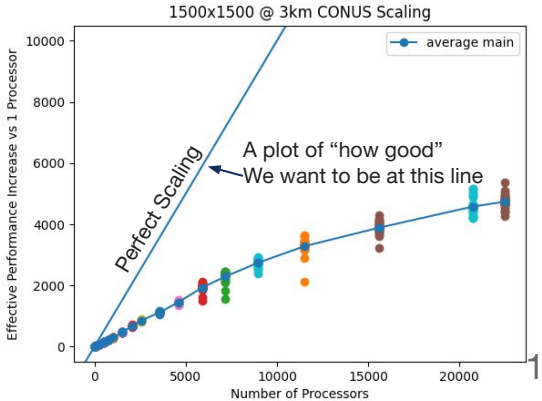
We will be using this style of plot from now on



← This looks as expected. A hyperbolic decay of timing almost characterized by 1/N

But this is difficult to view performance trends in a granular manner

This is a scaling plot → where we reinterpret the data with respect to some nominal value that constitutes perfect (100%) computational efficiency



Scaling - Explanation of Plot

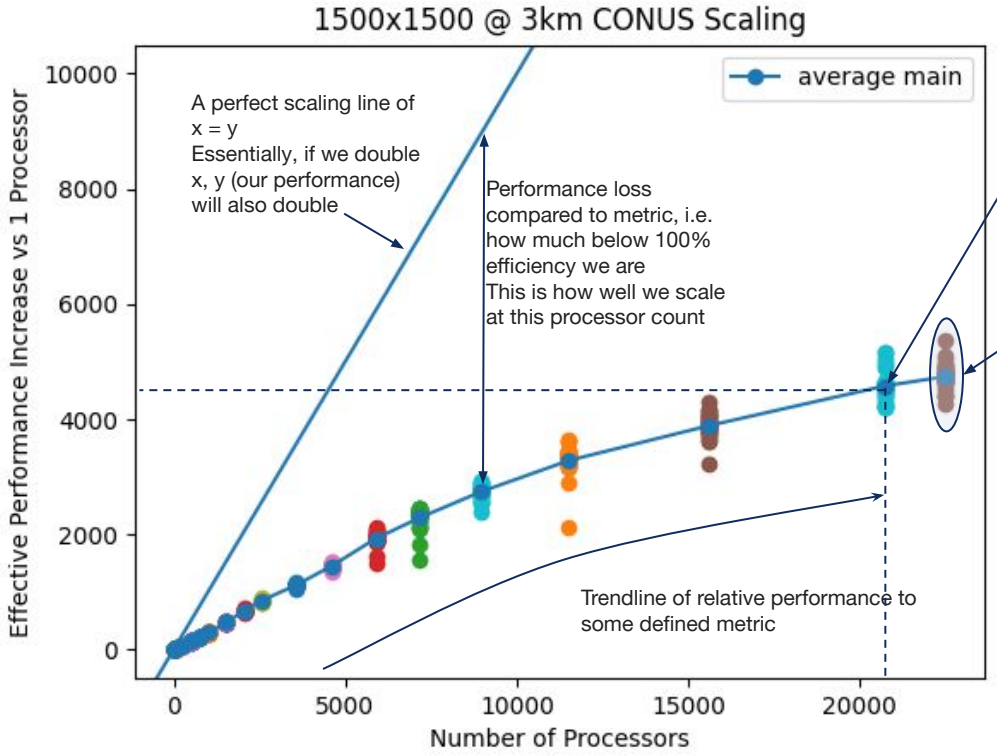
The metric we are comparing against.

The calculation for this is:

$$\frac{\text{(metric time) * (metric proc \#)}}{\text{recorded time}}$$

This formula reparametrizes the values such that a time step in the prescribed metric is operating at 100% efficiency, and for that specific metric the output will be 1 (i.e. 100% performance).

If 2x metric processors are used, an equivalent halving of the time to calculate the total simulation would be correctly represented as a 2, or 200% performance increase.



Average efficiency at this processor count is easy to calculate if 100% is x = y
For this point it would be :
4580 / 20736 => *22.09%*

The spread of values from simulation run

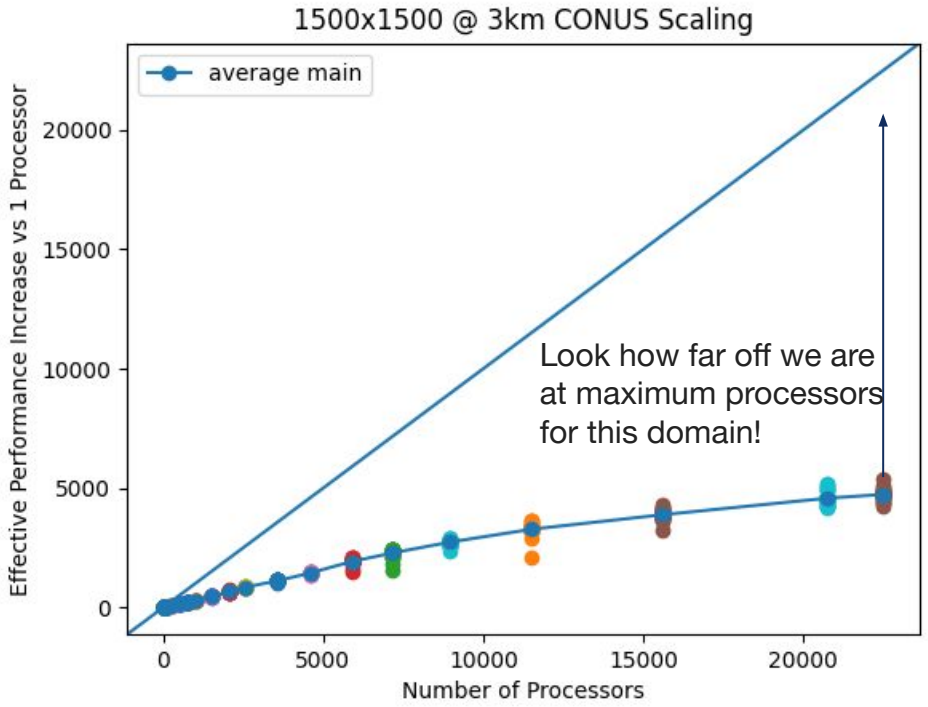
This is not shown in most future plots since we are focusing on average performance

Scaling - MPI Performance vs 1 Processor

When 1 Processor is used as a metric, as soon as we add parallelization scaling drops off dramatically.

This metric heavily penalizes *any* latency that is not directly progressing the simulation computation.

Using a metric that captures realistic latency would be a more reasonable real-world comparison when looking at massive parallelization anyways

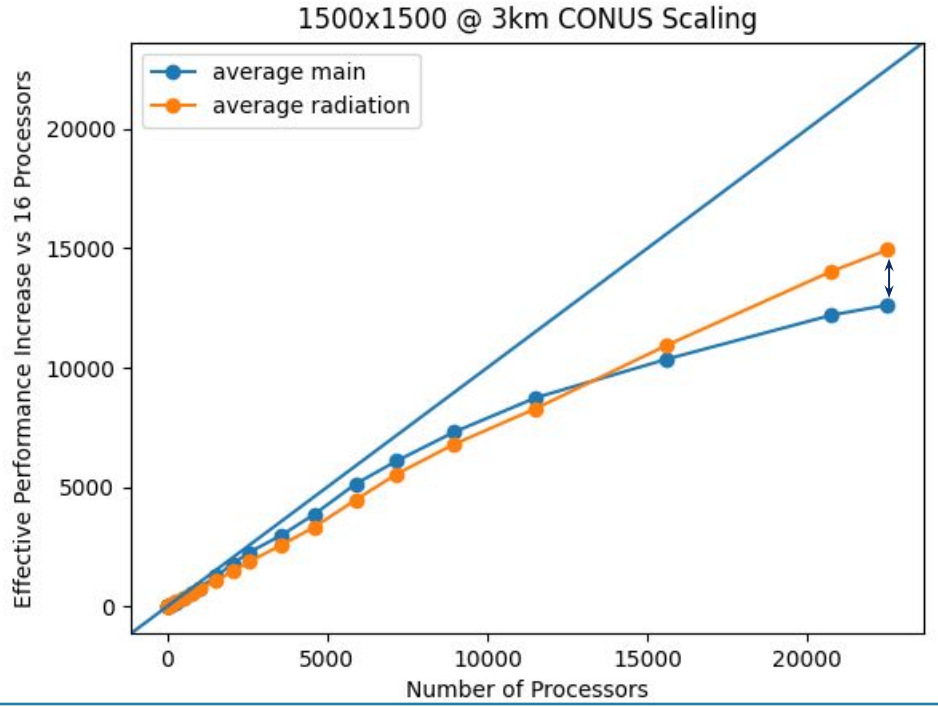


Scaling - MPI Performance vs 16 Processor

Using 16 MPI ranks as a baseline metric properly represents a nominal amount of parallelization overhead, but does not account for internode communications when scaling beyond 1 node.

At maximum 22500 processors :
52.65% for non-rad timesteps
66.90% for radiation timesteps

We can see that our radiation timesteps at larger processor counts continues to scale better than our normal timesteps. Radiation steps often take longer, so why might they appear to scale better?



Choosing a useful metric as a baseline is critical to presenting scaling as it is inherently relative

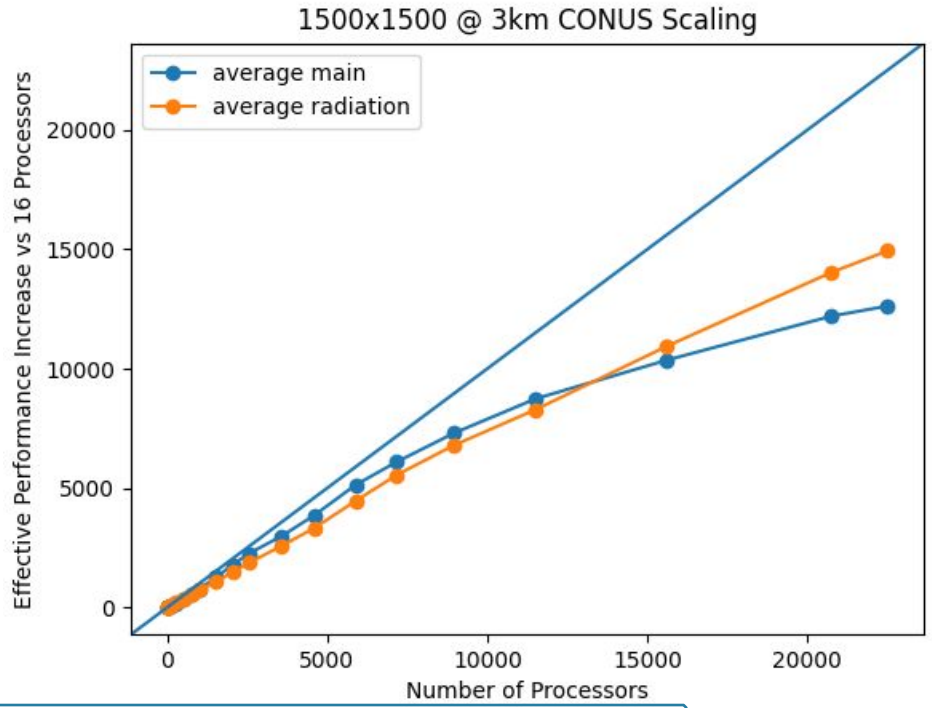
Scaling - MPI Performance vs 16 Processor

Recall the concepts learned from the **halo exchange** thought experiment :

Better computational efficiency is associated with a higher proportion of time spent doing the computation vs total time to perform the task.

Radiation time steps are computationally expensive and thus more time is spent on the **cell** computation than the parallel coordination in general.

This trend continues stronger than the analogous main timesteps that start to become more dominated by parallel overhead.



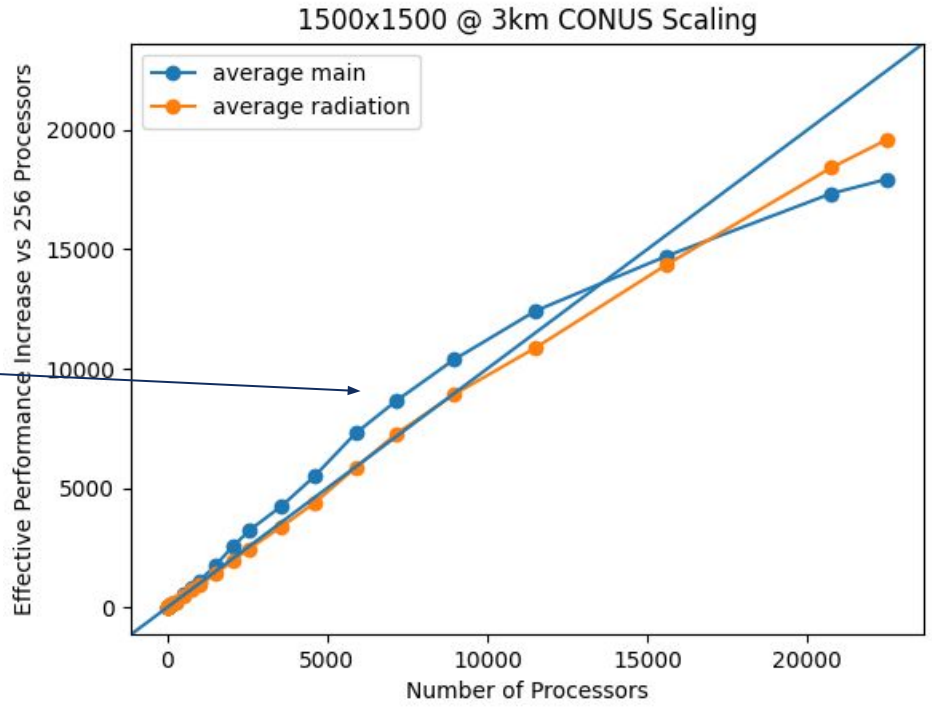
Computationally expensive tasks may generally have stronger scaling

Scaling - MPI Performance vs 256 Processor

To better characterize massive parallelization we might set our baseline to 256 processors for our use case. This makes total use of two nodes and captures the first latencies of internode communications.

But now we have our performance going beyond the “theoretical” limit of scaling. How?

This is referred to as **superlinear scaling**



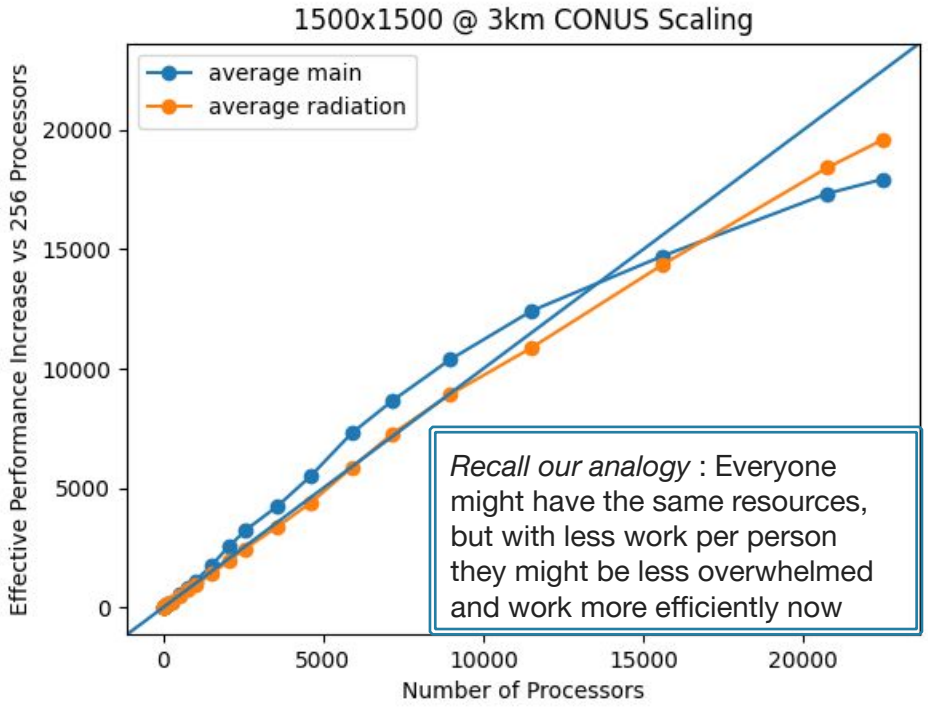
Scaling - MPI Performance vs 256 Processor

Recall that these are reinterpretations of performance to some baseline, and in these runs we are simply outperforming that baseline.

One might still ask:

- How are we outperforming that baseline?
- Wasn't 1 process the most efficient?
- All nodes are fully occupied, and all processes still get the same resources, so processes are not working with any more or less resources, right?

A possible explanation could be due to domain decomposition each process has more resources relative to the size of computation now, getting an effective performance boost in areas where computation was resource constrained



Scaling - MPI vs OMP vs 1 Processor (Single Node)

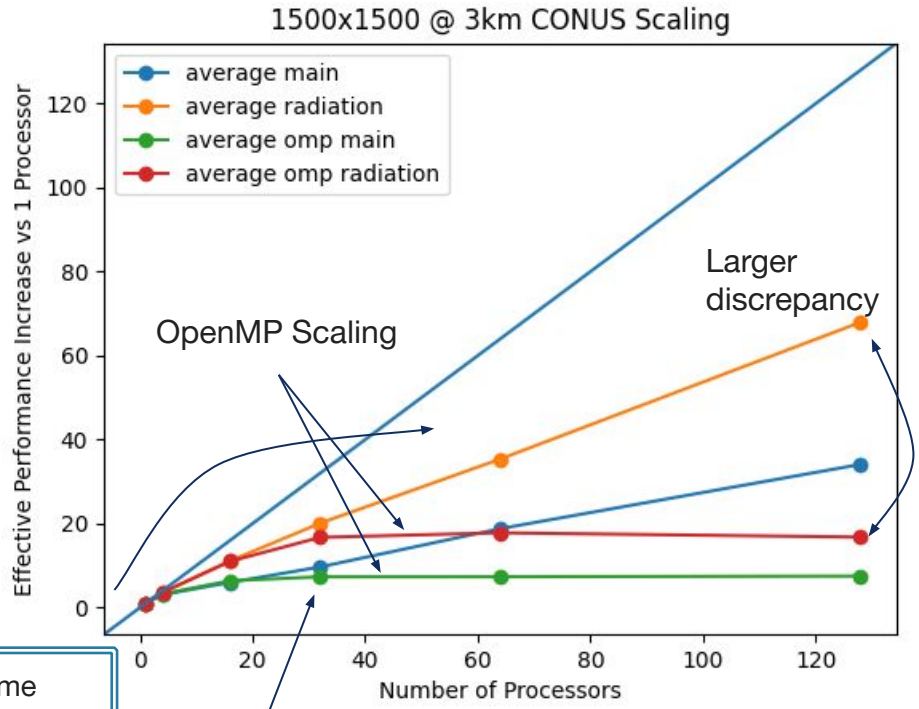
What does scaling look like within a node?

Less linear scaling within a node than MPI

- Possibly due to more coupled usage of computing resources when within the same shared memory space

Large discrepancy between radiation timesteps scaling

- Possibly due to the same thing that affords rad steps better scaling exacerbates MPI vs OpenMP
  - Computationally intensive with potentially many resources used
  - Threads within the same memory space might be thrashing each other



Recall our analogy : Once "full", adding more people to the same table is less effective than giving everyone a separate table even if the same total surface area is used to isolate workspaces

Scaling - Nested Domains

Processing of nested domains requires propagating forward 1 timestep in the parent beforehand to generate lateral boundary conditions

- Nested domain layers are inherently a serialized process
- All available cores are used to quickly go through a timestep
- Parent domains and nested domains use the same number of processors for domain decomposition
- Domain decomposition is limited to 10x10 **cell** sizes
- The cost of nested domain steps can easily dominate the total cost of a simulation

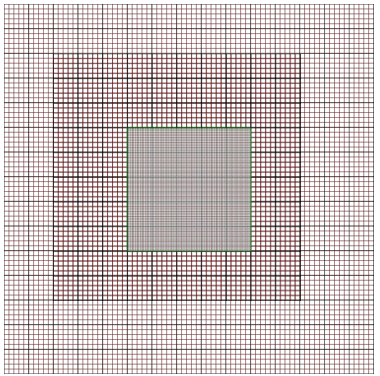
If the nested domain has a significantly higher number of **cells** than the parent domain, the parent domain will limit the number of processors able to be used for domain decomposition even though it is less computationally expensive!

Scaling - Nested Domains

Example: A 500x500 parent domain with a nested domain of size 250x250 *in parent grid* ***cells*** with a grid ratio of 3 and a timestep ratio of 3 as well - totaling 750x750 grid **cells**.

The nested domain would have a **cell** amount 2.25x more than the parent! Our parent domain is limited to a domain decomposition of 50x50, or 2500 processors, whereas our nested domain *would* be limited to 75x75, or 5625 processors.

If the parent domain computation takes only 13% of the total time, we would be throttling our scaling to something that only makes a minimal impact on our total runtime.



Keeping the nested domain the same size, if we increase the parent domain to size 750x750, we now have the same limit on scaling for both domains. This would make the parent domain take 25% of our total runtime, but now with the ability to add >2x processors

Try to make your nested and parent domain total ***cell*** *sizes at least match to maximize scaling of the two.*

Summary

Maximizing computational performance depends on many factors, including what performance metric you are trying to optimize - time vs efficiency.

Latency, especially overhead of memory access and parallel software coordination, play a fundamental role in how well software can be accelerated.

Due to the dynamics involved in NWP, as we continue to partition out smaller workloads via **domain decomposition** coordination of lateral boundary conditions - such as with **halo exchanges** - are a significant factor in limiting how well WRF can scale.

Understanding scaling works best if one understands the baseline metric used to scale against and the implications of that chosen scale.

Computationally expensive tasks may exhibit stronger overall scaling as the overhead of parallelization is not as prevalent as workload decreases.

Using a scaling graph, we can better understand how to balance expected runtime and performance increase to our available computing resources.

When seeking to maximize scaling of a nested domain, consider the limitations the parent domain has and appropriately scale the parent to allow the nested domain to reach peak scaling.

Thank you

Backup slides

MPI Timings 1500x1500x50, 3km, CONUS Suite vs 256 proc

Processor Count	i dim	j dim	Main(s)	Rad(s)	Main Perf	Rad Perf	Main Scaling	Rad Scaling
1	1500	1500	248.24584	1123.70343	3.78431	1.88983	3.78431	1.88983
4	750	750	80.91518	311.41487	11.61019	6.81923	2.90255	1.70481
16	375	375	41.32153	101.12227	22.73489	21.00042	1.42093	1.31253
32	375	188	25.64082	55.92232	36.63847	37.97429	1.14495	1.1867
64	188	188	13.20109	31.75976	71.16385	66.86482	1.11194	1.04476
128	188	94	7.27732	16.53463	129.09157	128.43413	1.00853	1.00339
256	94	94	3.66969	8.29535	256	256	1	1
512	94	47	1.68862	4.19213	556.33762	506.57116	1.0866	0.9894
768	63	47	1.11251	2.74242	844.43655	774.35733	1.09953	1.00828
1024	47	47	0.8336	2.19742	1126.97118	966.41228	1.10056	0.94376
1536	47	32	0.52214	1.49019	1799.19726	1425.0602	1.17135	0.92777
2048	47	24	0.36948	1.08746	2542.61475	1952.81109	1.24151	0.95352
2560	38	24	0.29179	0.86425	3219.60759	2457.162	1.25766	0.95983
3584	27	24	0.22073	0.62501	4256.04498	3397.72236	1.18751	0.94803
4608	24	21	0.17076	0.48489	5501.47099	4379.57156	1.1939	0.95043
5888	24	17	0.12879	0.36253	7294.12193	5857.80481	1.23881	0.99487
7168	24	14	0.10844	0.2916	8663.24937	7282.61472	1.2086	1.01599
8960	19	14	0.09053	0.23812	10377.24385	8918.36125	1.15817	0.99535
11520	16	13	0.07559	0.19506	12427.41864	10886.96018	1.07877	0.94505
15616	13	12	0.0638	0.14779	14725.28336	14369.43197	0.94296	0.92017
20736	11	11	0.05421	0.11534	17330.805	18411.74314	0.83578	0.88791
22500	10	10	0.05237	0.10833	17939.37758	19602.55803	0.79731	0.87122

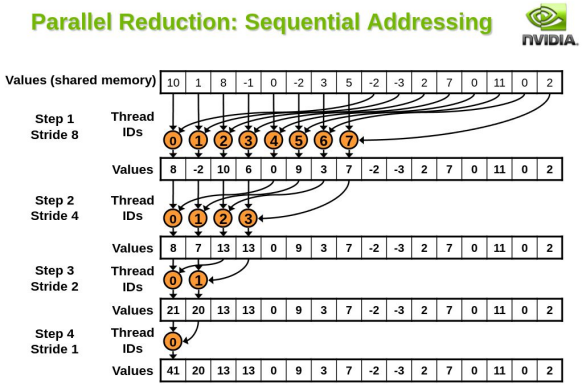
Typical Usage / Techniques

Using the concepts built up, we can better understand typical parallelization techniques :

- Independent tasks for reading/writing to “disk” (isolate slowest memory accesses)
- Load data in chunks, and localize common data to similar regions for quick access
- Defer synchronization/coordination events until necessary and group these events together
- Where possible make tasking instructions (software) generally the same and independent for bulk repeating tasks as these can then be split up without impacting other tasking

Splitting tasking to max the amount of work each worker does without stopping for other tasks can vastly increase our real-world time cost.

Each reduction step is a synchronization + is mindful of the location of memory



NVIDIA : Optimizing Parallel Reduction in CUDA