Parallel Computing Concepts

Robert Sinkovits, PhD
(retired) Director of Education
San Diego Supercomputer Center

https://github.com/sdsc-complecs/Parallel-computing-concepts

Table of contents

Introduction

The reason most of you are here is that your machine learning (ML) workloads have grown to the point where you can no longer run them on your local resources (e.g., laptops, desktop systems). Specifically, you may need to use parallel computing.

This presentation was adapted from a more general talk on parallel computing concepts, but the ideas still apply in the context of ML.

This session is intended for anyone who …

- Currently runs, plans to run or is thinking about running applications on parallel computers
- Writes proposals for computer time on campus clusters, nationally allocated systems or other resources
- Purchases time on compute resources and wants to maximize the return on their investment
- Is considering purchasing hardware for their lab
- Is simply curious about parallel computing

Motivation

- Much of the training in parallel computing is targeted at people who write their own parallel applications and focuses on programmer topics (MPI, OpenMP, CUDA, profiling, performance tuning).
- Consequently, end users who are not developers rarely or never get a proper introduction to parallel computing.
- Even if you don't write code, it's still important to understand some of the basic principles of parallel computing so that you make the most effective use of valuable advanced cyberinfrastructure.

Parallel computing myths

Myth #1: Parallel computing is for astrophysicists, engineers, climate modelers and others working in traditionally math intensive fields

While this might have been (partially) true decades ago, today nearly every field of research makes use of parallel computing. This includes the social sciences, life sciences, arts & humanities in addition to physics, chemistry, engineering, material science and the other “usual suspects”.

Parallel computing myths

Myth #2: Throwing more hardware at a problem will automatically reduce the time to solution

Parallel computing will only help if you have an application that has been written to take advantage of parallel hardware. And even if you do have a parallel code, there is an inherent limit on scalability.

Caveat – a high-throughput computing workload can use parallel computing to run many single-core/GPU instances of an application to achieve near perfect scaling.

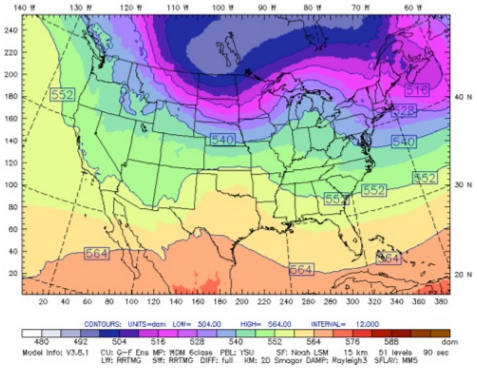
Parallel computing myths

Myth #3: You need to be a programmer or software developer to make use of parallel computing

Most users of parallel computers are not programmers. Instead they use mature 3rd party software that had been developed elsewhere and made available to the community.

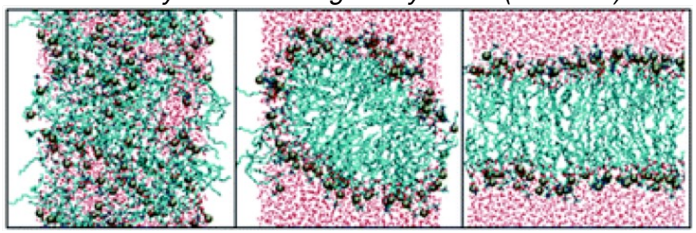
Many of you will be using someone else's code(s)

*Weather modeling (**WRF**)*



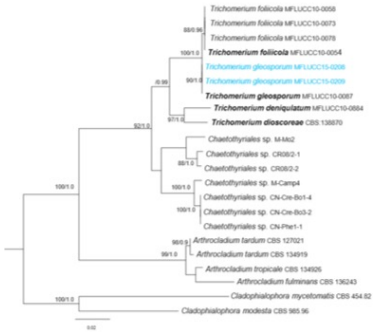
https://www2.mmm.ucar.edu/projects/wrf-model/plots/realtime_main.php

*Molecular dynamics biological systems (**AMBER**)*



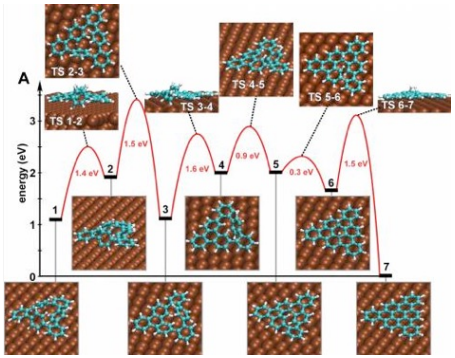
Dickson et. al, Soft Matter (2012)

*Phylogenetics (**RAxML**)*



Hongsanan et al., Mycosphere (2016)

*Electronic structure (**CP2K**)*



https://www.cp2k.org/exercises:2015_cecam_tutorial:neb

Parallel computers

Modern clusters and parallel computers consist of multiple compute nodes connected by a fast network. Each CPU node contains one or more (typically two) multicore processors, while GPU nodes usually contain four GPUs

To effectively use this hardware, we need applications that have been parallelized so that they can run on multiple cores/GPUs (within a node) or across multiple nodes.

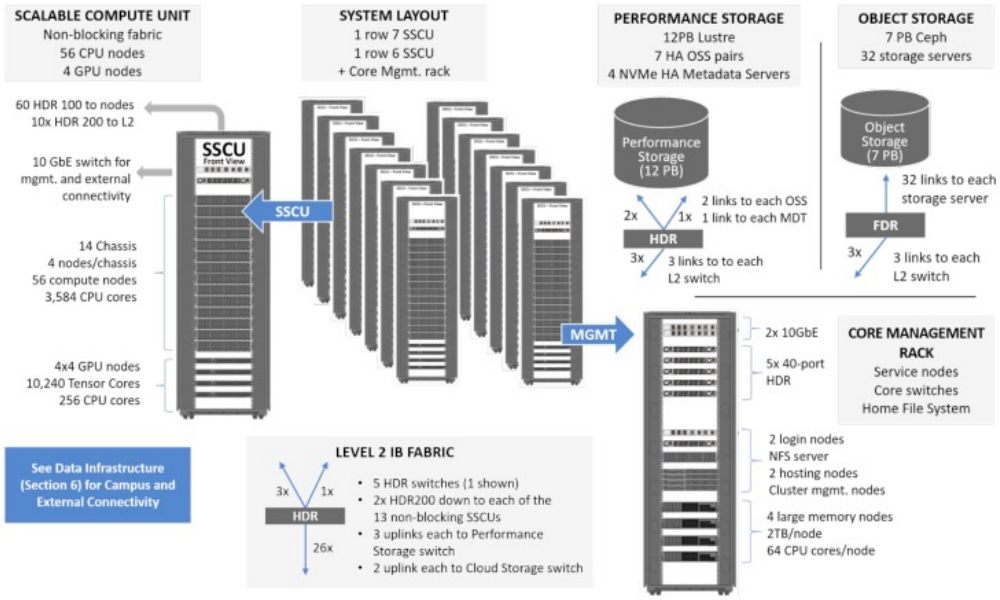


Table of contents

Processes and threads

Threads and processes are both independent sequences of execution

- **Process:** instance of a program, with access to its own memory, state and file descriptors (can open and close files)
- **Thread:** lightweight entity that executes inside a process. Every process has at least one thread and threads within a process can access shared memory.

Online resources describing the differences between threads and processes tend to be geared toward computer scientists, but the following resources do a reasonable job of addressing the topic

- https://stackoverflow.com/questions/200469/what-is-the-difference-between-a-process-and-a-thread (nerdier)
- https://www.educba.com/process-vs-thread/ (more informal)

Processes and threads

- **Processes**
  - Incur more overhead
  - Are more flexible – multiple processes can be run within a compute node or across multiple compute nodes (distributed memory)
- **Threads**
  - Incur less overhead
  - Threaded codes can use less memory since threads within a process have access to the same data structures
  - Are less flexible – multiple threads associated with a process can only be run within a compute node (shared memory)

Processes and threads – why do I care?

The type of parallelization will determine how/where you will run your code.

- Distributed-memory applications (multiple processes/instances of a program) can be run on one or more nodes
- Shared-memory (threaded) applications should be run on a single node
- Hybrid applications can be run on one or more nodes, but should consider the balance between threads and processes (discussed later)
- In all cases, may need to consider how processes and threads are mapped and bound to cores

In addition, being aware of threads and processes will help you to understand how your code is utilizing the hardware and identify common problems.

Message Passing Interface (MPI)

MPI is a standard for parallelizing C, C++ and Fortran code to run on distributed memory (multiple compute node) systems. While not officially adopted by any major standards bodies, it has become the *de facto* standard (i.e., almost everyone uses it).

- There are multiple open-source implementations, including OpenMPI, MVAPICH, and MPICH along with vendor-supported versions.
- MPI applications can be run within a shared-memory node. All widely-used MPI implementations are optimized to take advantage of the faster intranode communications.
- MPI is portable and can be used anywhere.
- Although MPI is often synonymous with distributed memory parallelization, other options are gaining adoption (Charm++, UPC, X10)

Message Passing Interface (MPI)

Although we're discussing MPI, the same principles apply when parallelizing deep learning applications using …

- Horovod – a distributed deep learning training framework for TensorFlow, Keras, PyTorch, and Apache MXNet
- NCCL – The NVIDIA Collective Communication Library (NCCL), which implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and Networking

MPI example – Hello World

```
#include <stdio.h>

int main(int argc, char** argv) {
   // Print off a hello world message
   printf("Hello world \n")
}
```

MPI applications are pretty dense and written at a low level. Data is explicitly communicated between processes using calls to MPI library routines.

If you are not a programmer or application developer, you do not need to know MPI. You should just be aware that it exists and that if you are building your executable, you'll need to use the appropriate wrappers.

MPI example – Hello World

```
#include <mpi.h>
#include <stdio.h>

int main(int argc, char** argv) {
   // Initialize the MPI environment
   MPI_Init(NULL, NULL);

   // Get the number of processes
   int world_size;
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);

   // Get the rank of the process
   int world_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

   // Get the name of the processor
   char processor_name[MPI_MAX_PROCESSOR_NAME];
   int name_len;
   MPI_Get_processor_name(processor_name, &name_len);

   // Print off a hello world message
   printf("Hello world from processor %s, rank %d out of %d processors\n",
          processor_name, world_rank, world_size);

   // Finalize the MPI environment.
   MPI_Finalize();
}
```

MPI applications are pretty dense and written at a low level. Data is explicitly communicated between processes using calls to MPI library routines.

If you are not a programmer or application developer, you do not need to know MPI. You should just be aware that it exists and that if you are building your executable, you'll need to use the appropriate wrappers.

Message Passing Interface (MPI) insight

Although MPI can be used to integrate multiple applications, most often you will use it to parallelize a single application

In that case, you will be running N independent instances of the same program, each of which is initially distinguished solely by its rank (0 ≤ rank ≤ N)

The rank will determine things such as

- Portion of the problem to work on
- File or pointer to a file specifying data to be read or written
- Other processes to communicate data with

OpenMP

OpenMP is an application programming interface (API) for shared-memory (within a node) parallel programming in C, C++ and Fortran

- OpenMP provides a collection of compiler directives, library routines and environment variables.
- OpenMP is supported by all major compilers, including IBM, Intel, GCC, PGI, and AMD Optimizing C/C++ Compiler (AOCC).
- OpenMP is portable and can be used anywhere.
- Although OpenMP is often synonymous with shared-memory parallelization, there are other options: Cilk, POSIX threads (pthreads) and specialized libraries for Python, R and other programming languages.

OpenMP example – loop parallelization

```
main ()  {

int i;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;

for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

Applications parallelized using OpenMP tend to be more readable since they often involve relatively minor changes to the code.

If you are not a programmer or application developer, you do not need to know OpenMP. You should just be aware that it exists and that if you are building your executable, you'll need to use the appropriate compiler flags (e.g., -fopenmp, -qopenmp or -mp)

OpenMP example – loop parallelization

```
#include <omp.h>
#define N 1000
#define CHUNKSIZE 100

main () {

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel for \
     shared(a, b, c, chunk) private(i) \
     schedule(static, chunk)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
}
```

Applications parallelized using OpenMP tend to be more readable since they often involve relatively minor changes to the code.

If you are not a programmer or application developer, you do not need to know OpenMP. You should just be aware that it exists and that if you are building your executable, you'll need to use the appropriate compiler flags (e.g., -fopenmp, -qopenmp or -mp)

MPI and OpenMP big picture

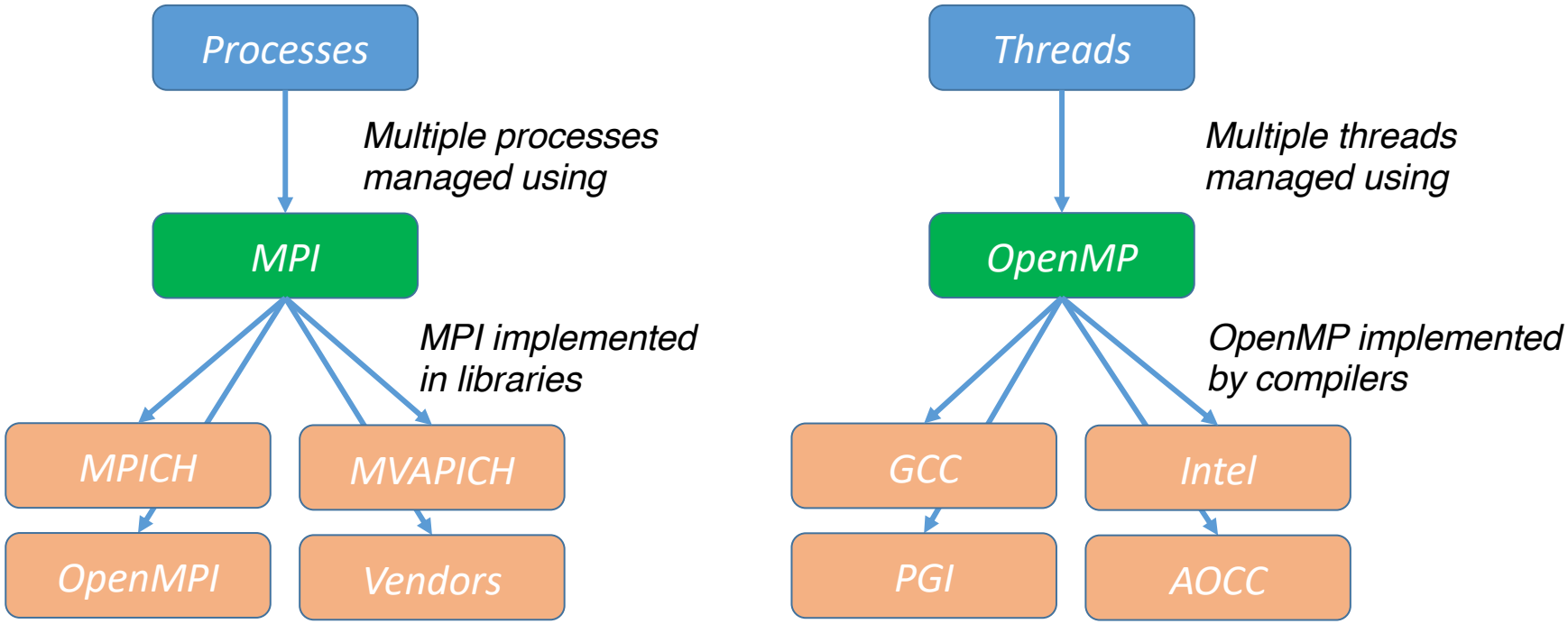


Table of contents

Hybrid applications

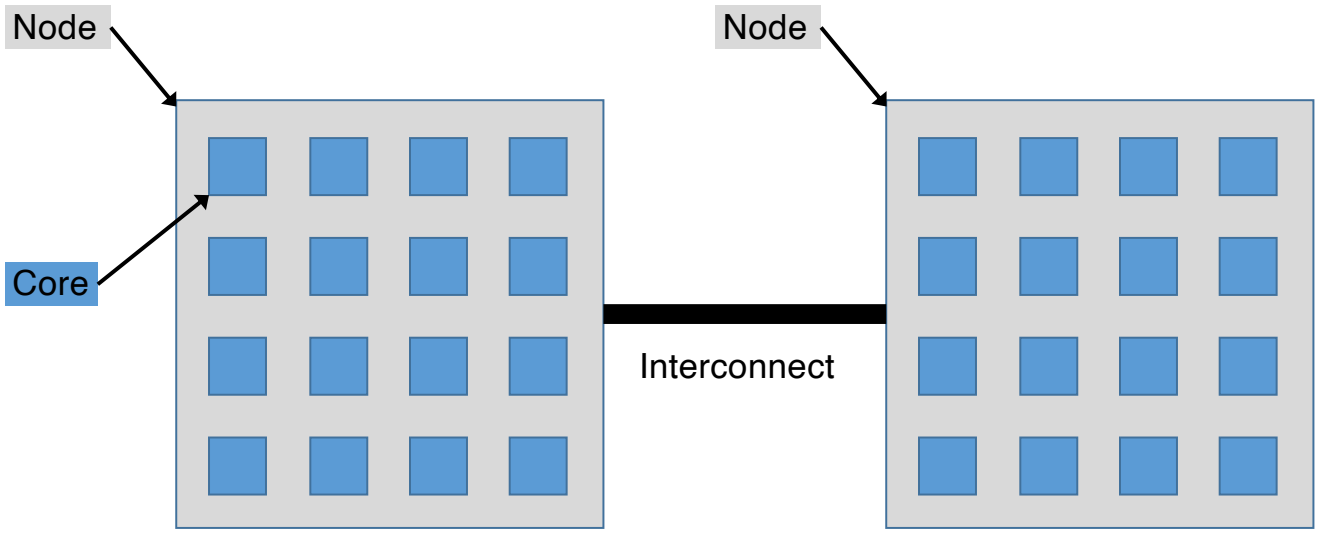
Many modern parallel applications are built using a hybrid approach to take advantage of both distributed and shared memory. This typically involves MPI and OpenMP, although other combinations are possible.

Hybrid codes have advantages over purely shared or distributed memory apps

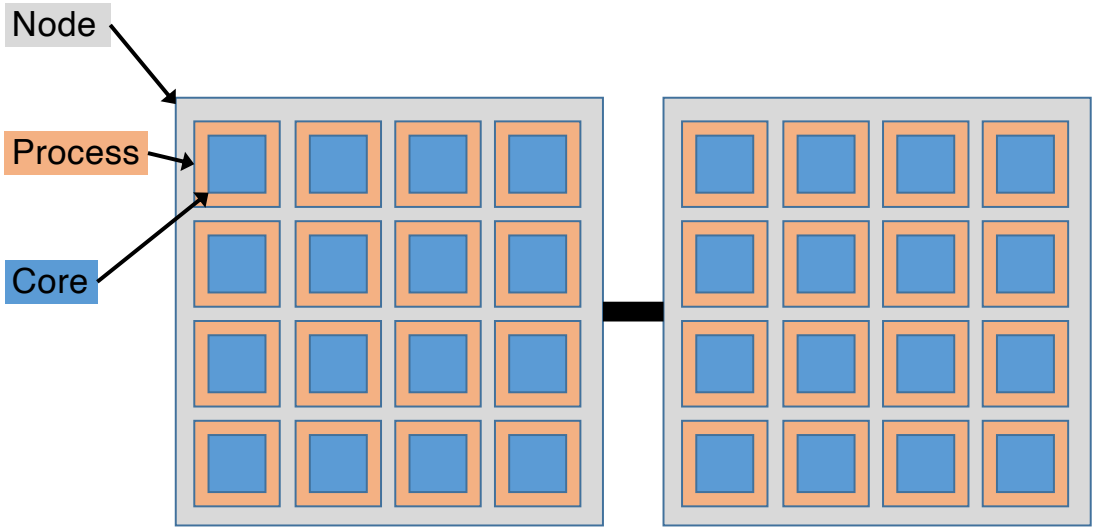
- Shared memory apps can have limited scalability within a node and cannot be run across multiple nodes
- Distributed memory apps may have higher memory requirements and more overhead when running within a node

Simplified parallel computer

To illustrate running parallel applications, we'll consider a simplified system consisting of two nodes, each with 16 cores, joined by a network.



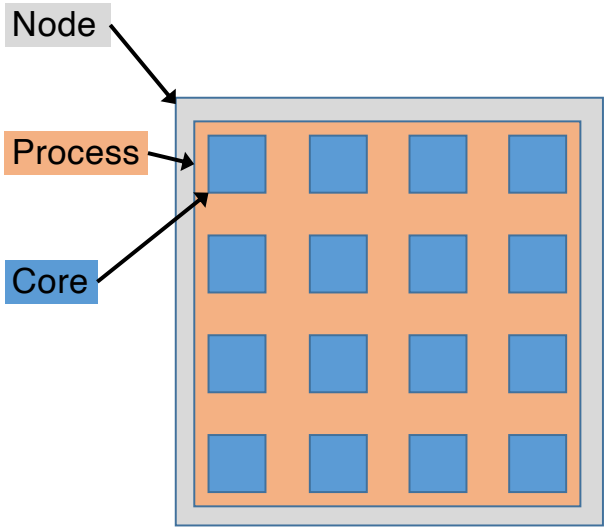
Message passing applications



Applications that have been parallelized using message passing only are typically executed with one process per core.

This example shows an application running 32 processes across two 16-core nodes.

Threaded applications

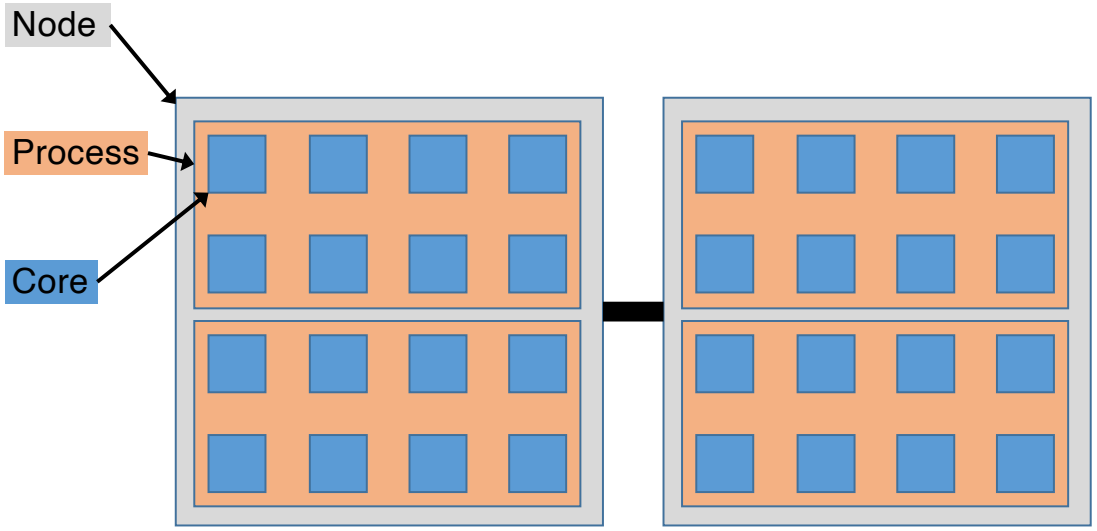


Applications that have been parallelized using threading only are restricted to running within a single node*

This example shows a threaded application (single process) running 16 threads on 16 cores

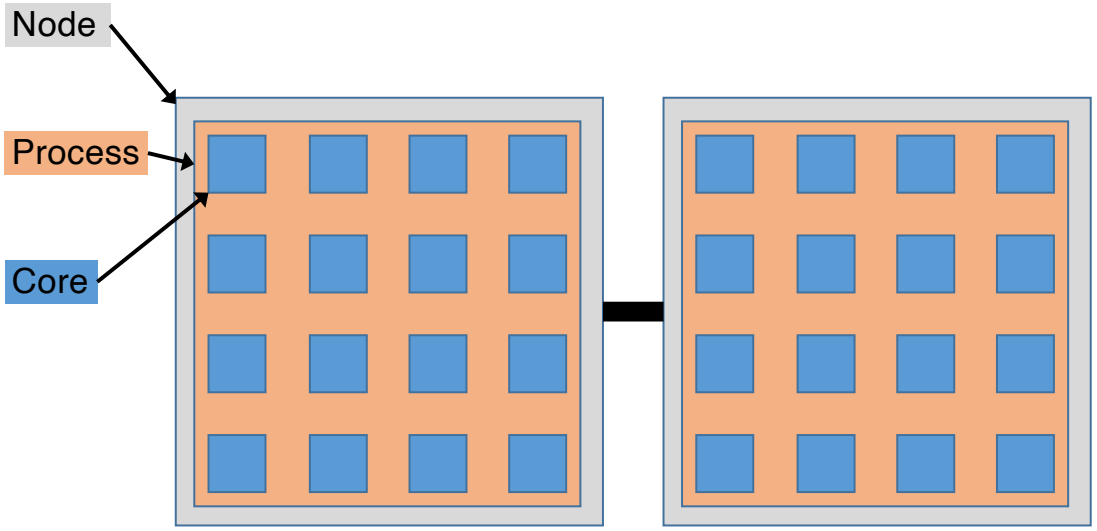
* Technically any programming model can be mapped to any hardware, but in practice threaded apps are run within a single node for best performance

Hybrid applications (typically MPI + OpenMP)



Any combination of threads and processes within a node is allowed. This example shows two processes per node and eight threads per process.

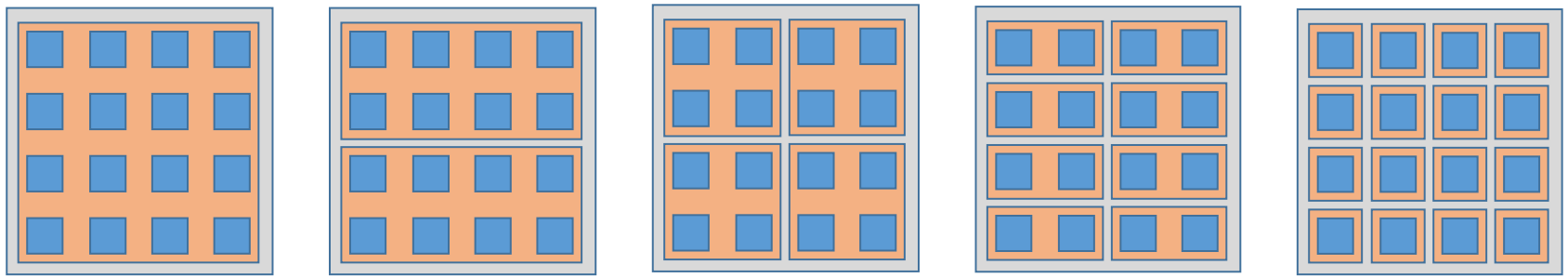
Hybrid applications (typically MPI + OpenMP)



A typical scenario is to run one process per node and use threads within a node, although this may become less common as the number of cores on a node continues to grow.

Hybrid apps – choosing balance between cores and threads

For hybrid applications, how do we choose the balance between processes and threads given the range of options from one process/node to one process/core? Need to take into account performance (run time) and memory usage.



1 process/node < ---------- > 1 process/core

Table of contents

Amdahl's law and limits on scalability

Amdahl's law describes the absolute limit on the speedup of a code as a function of the proportion of the code that can be parallelized and the number of processors. This is the most fundamental law of parallel computing!

- P is the fraction of the code that can be parallelized
- S is the fraction of the code that must be run sequentially (S = 1– P)
- N = number of processors

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

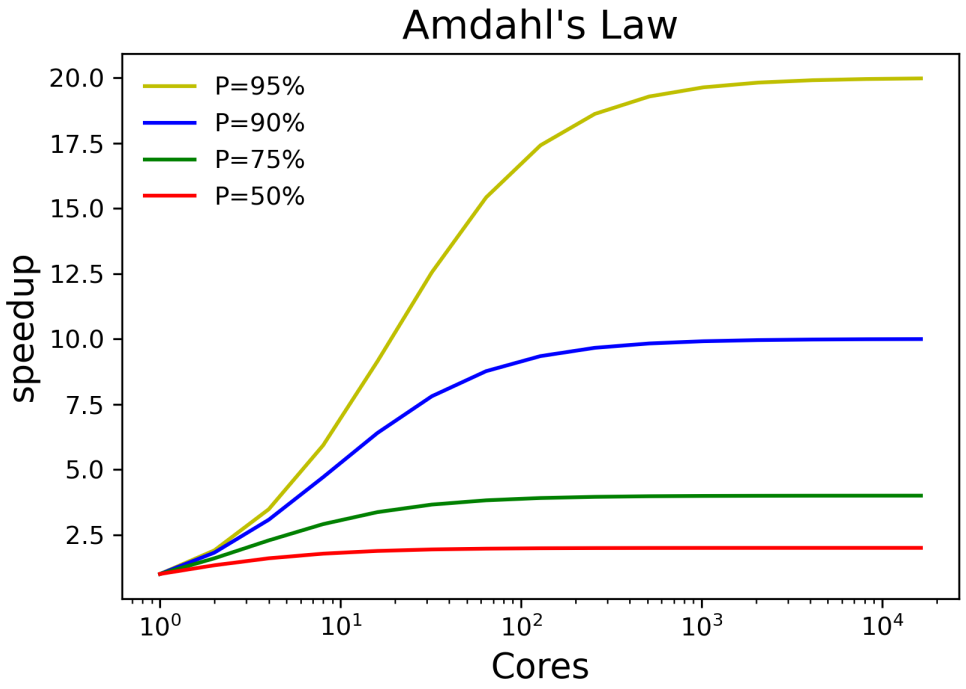
Amdahl's law and limits on scalability

In the limit as the number of processors goes to infinity, the theoretical speedup depends only on the proportion of the serial content

$$\lim_{N\to\infty} \frac{1}{(1-P)+P/N} = \frac{1}{(1-P)} = \frac{1}{S}$$

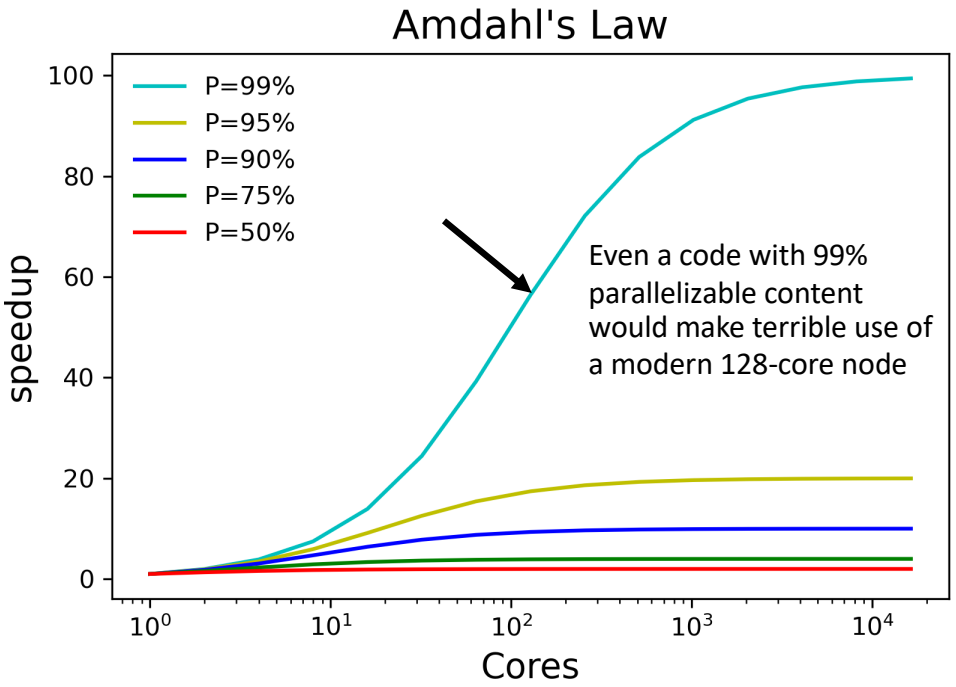
That doesn't look so bad, but as we'll show in the next slide it doesn't take much serial content to quickly impact the speedup

Amdahl's law and limits on scalability



P	Max speedup
0.50	2
0.75	4
0.90	10
0.95	20

Amdahl's law and limits on scalability



P	Max speedup
0.50	2
0.75	4
0.90	10
0.95	20
0.99	100

Amdahl's law – parallel and serial content

We talked briefly about serial and parallel content in a very circular way

- Parallel content consists of operations that could be done in parallel
- Serial content consists of operations that can't be parallelized

Let's try to make that a little more concrete by considering a simple example where we read a large list of numbers, apply a function *f* to each number, sum over the results and then apply a function *g* on the sum.

$$y = g\left(\sum_{i=1}^{N} f(x_i)\right)$$

Amdahl's law – parallel and serial content

If we were performing this computation on a single CPU, everything would be done serially – read data; apply the function f() to all elements of the list and accumulate results; apply g() to the sum. In the next few slides, we'll explore how some of this work can be done in parallel.

Read data

$$y = \sum_{i=1}^{i=N} f(x_i)$$

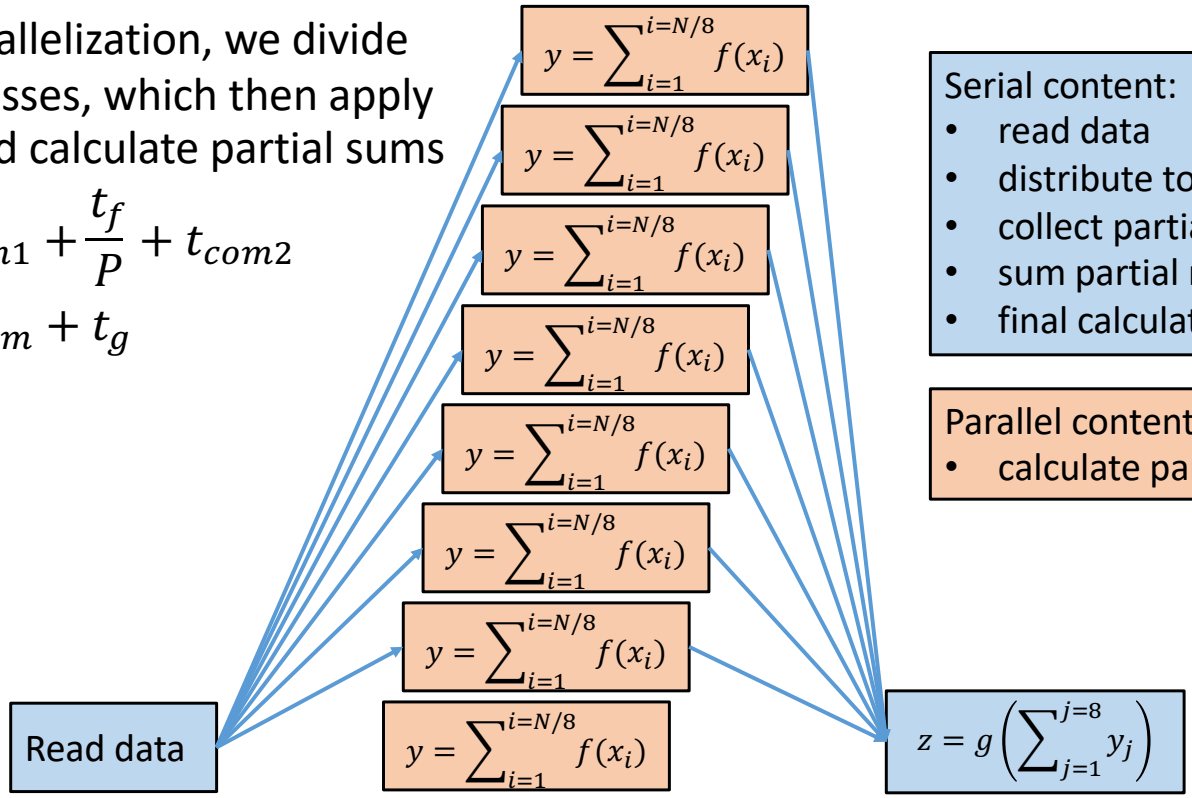
$$z = g(y)$$

Serial content:
- read data
- calculate sum over f(x)
- final calculation g(y)

Amdahl's law – parallel and serial content

In first attempt at parallelization, we divide the list across *P* processes, which then apply f() to their portion and calculate partial sums

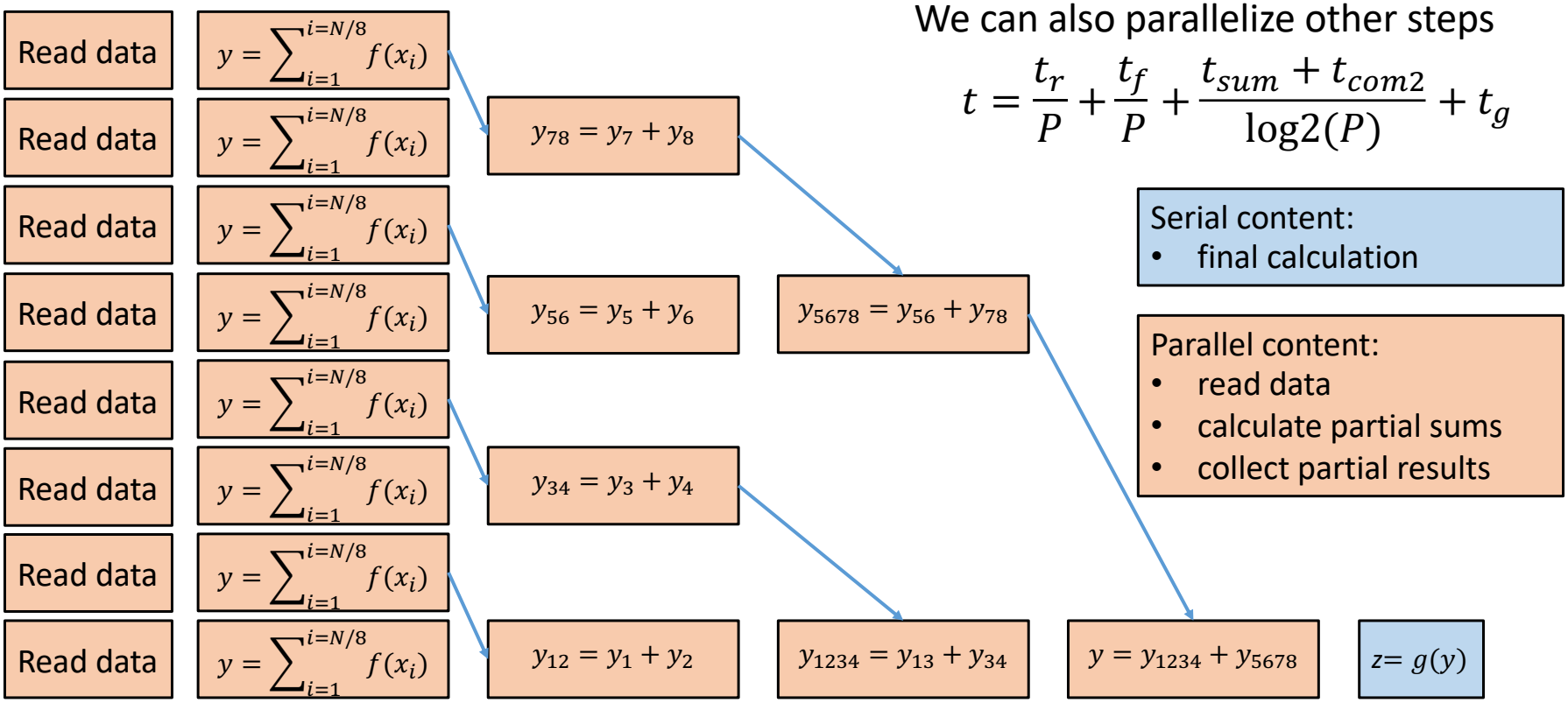
$$t = t_r + t_{com1} + \frac{t_f}{P} + t_{com2} + t_{sum} + t_g$$



Serial content:
- read data
- distribute to processes
- collect partial results
- sum partial results
- final calculation

Parallel content:
- calculate partial sums

Amdahl's law – parallel and serial content



We can also parallelize other steps

$$t = \frac{t_r}{P} + \frac{t_f}{P} + \frac{t_{sum} + t_{com2}}{\log2(P)} + t_g$$

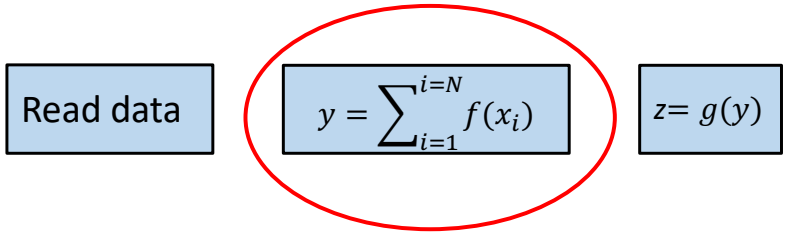
Serial content:
- final calculation

Parallel content:
- read data
- calculate partial sums
- collect partial results

Amdahl's law – parallel and serial content

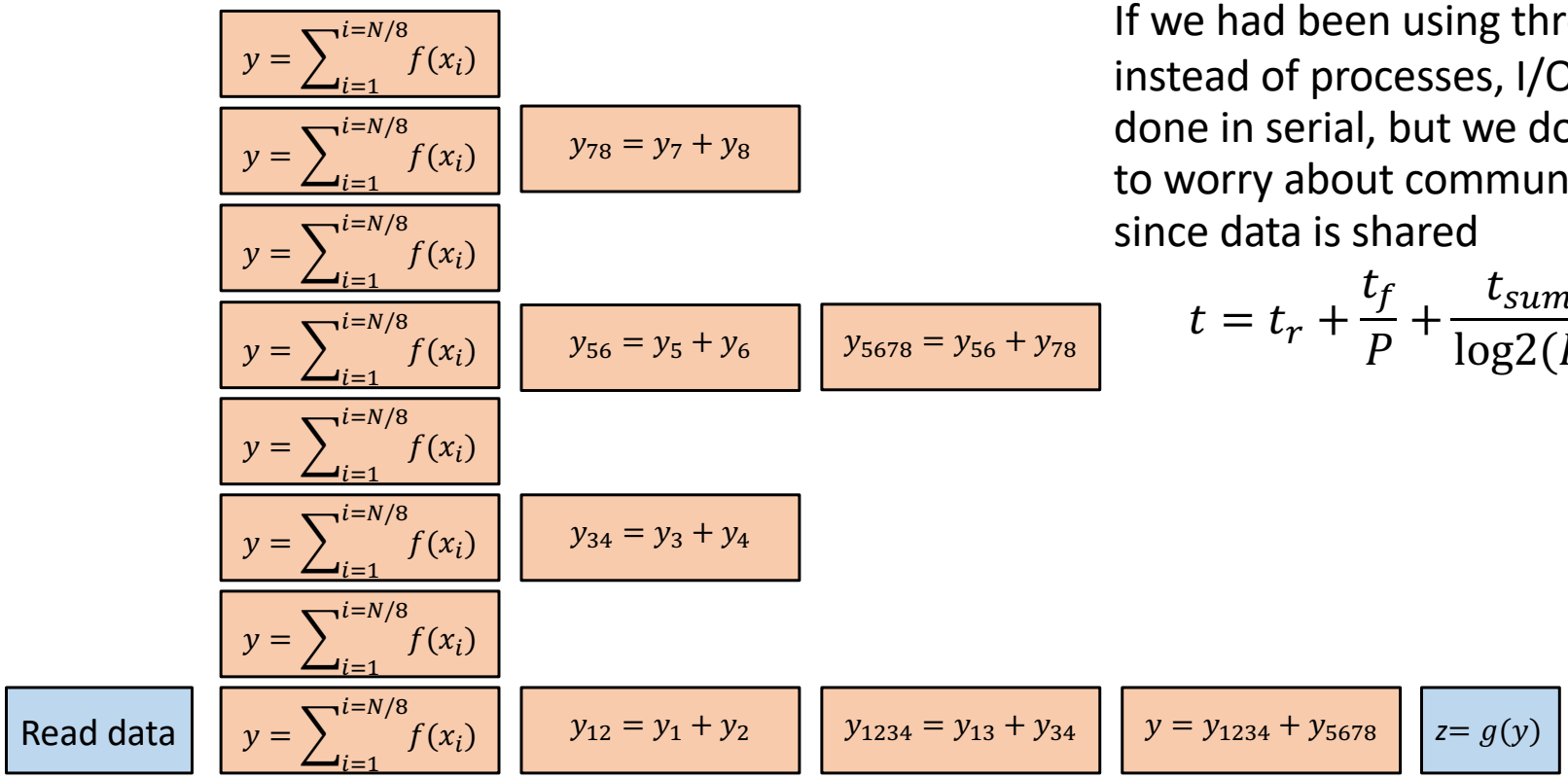
How far we go with the parallelization will depend on how much time is spent in each step. If most of the effort is in calculating f(x), with I/O and other costs are minimal, we might go with the first parallelization scheme.

Note that all robust implementations of MPI do collective operations (e.g., broadcast and reduction) of data using binary tree algorithms, so you won't need to implement that yourself.



Serial content:
- read data
- calculate sum over f(x)
- final calculation g(y)

Amdahl's law – parallel and serial content



If we had been using threads instead of processes, I/O will be done in serial, but we don't need to worry about communications since data is shared

$$t = t_r + \frac{t_f}{P} + \frac{t_{sum}}{\log 2(P)} + t_g$$

Table of contents

Other limits on scalability

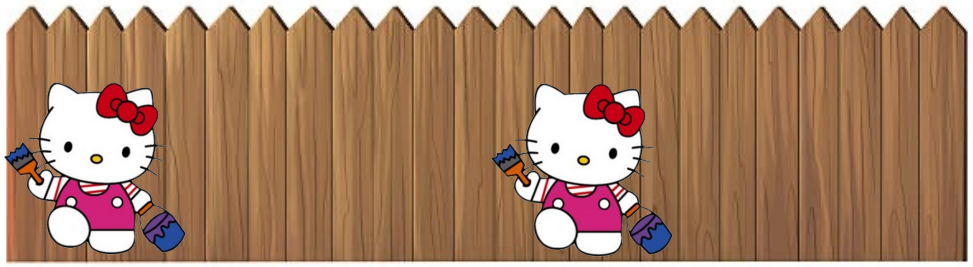
Amdahl's law sets a **theoretical upper limit** on speedup, but there are other factors that affect scalability:

- Problem size
- Communications overhead
- Uneven load balancing

In real-life applications that involve communications, synchronization (all threads or processes must complete their work before proceeding) or irregular problems (non-cartesian grids), the speedup can be much less than predicted by Amdahl's law.

Problem size limitations

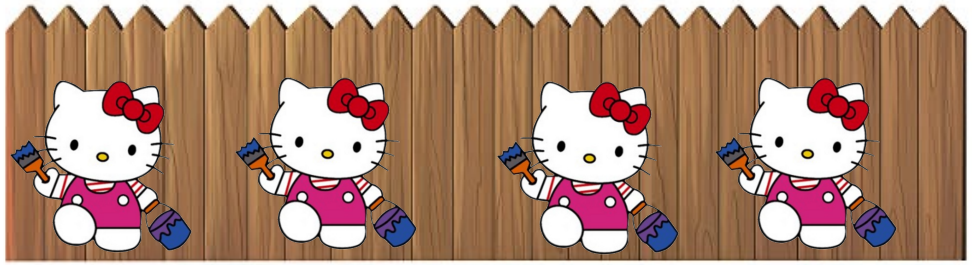
*One "Hello Kitty" takes 48 minutes to paint fence **



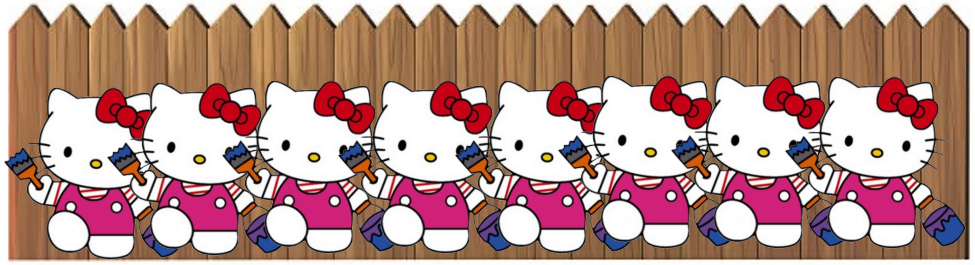
Two "Hello Kitty" take 24 minutes to paint fence

*** This was the only public domain image of a character with a paintbrush I could find*

Problem size limitations

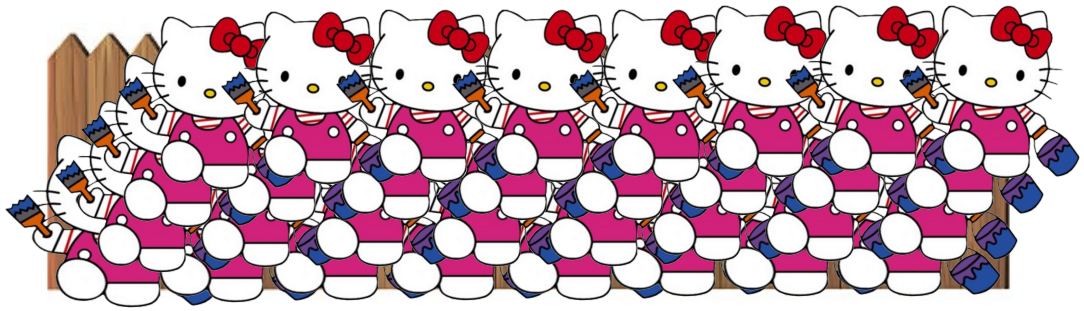


Four "Hello Kitty" take 12 minutes to paint fence



Eight "Hello Kitty" take 6 minutes to paint fence? Doubtful since it's getting a little crowded

Problem size limitations

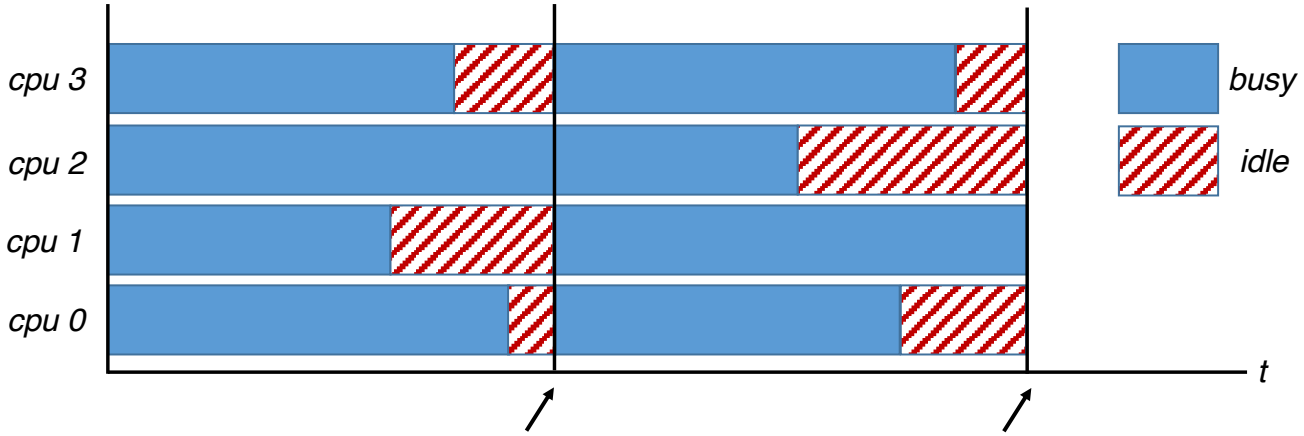


There is certainly not enough work for 32 “Hello Kitty” to paint the fence

Not only will you eventually get to the point where there’s not enough work to keep all your processors/GPUs busy, but you may see your application run slower since the parallel overhead overwhelms the run time

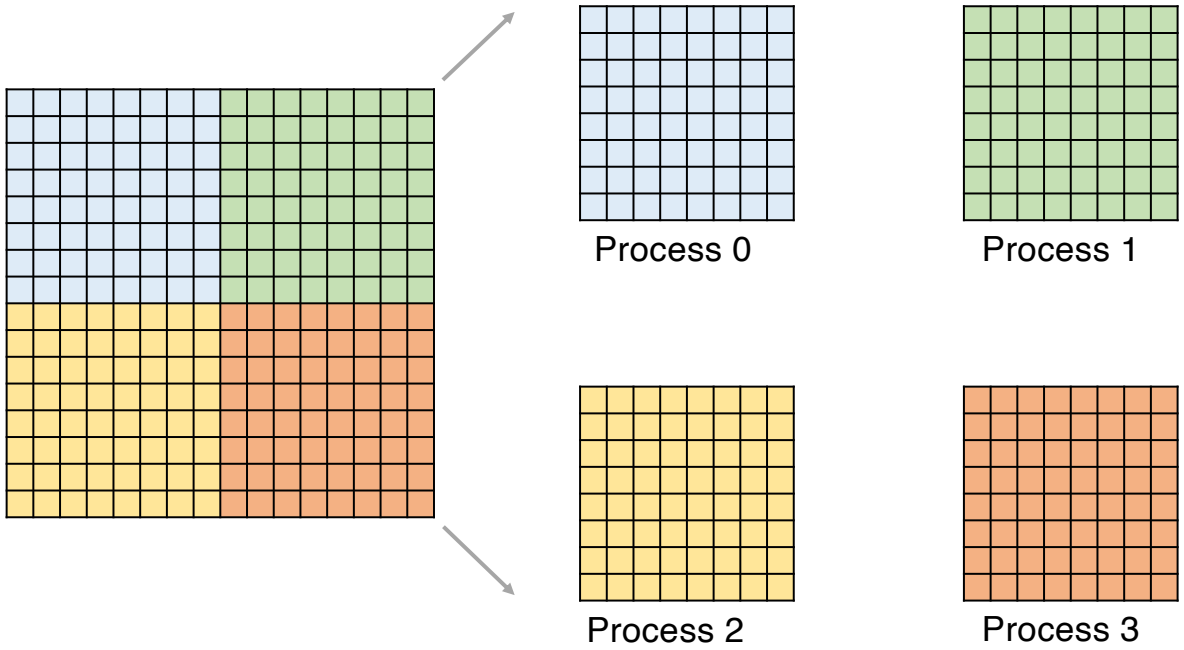
Uneven load balancing

Parallelizing a code requires dividing the computational work into chunks that can be executed independently. If the work cannot be distributed evenly, then processors will sit idle waiting for the longest chunk to finish.



synchronization points where all tasks must complete before proceeding (e.g., processes must exchange newly computed results or threads must finish updating shared array)

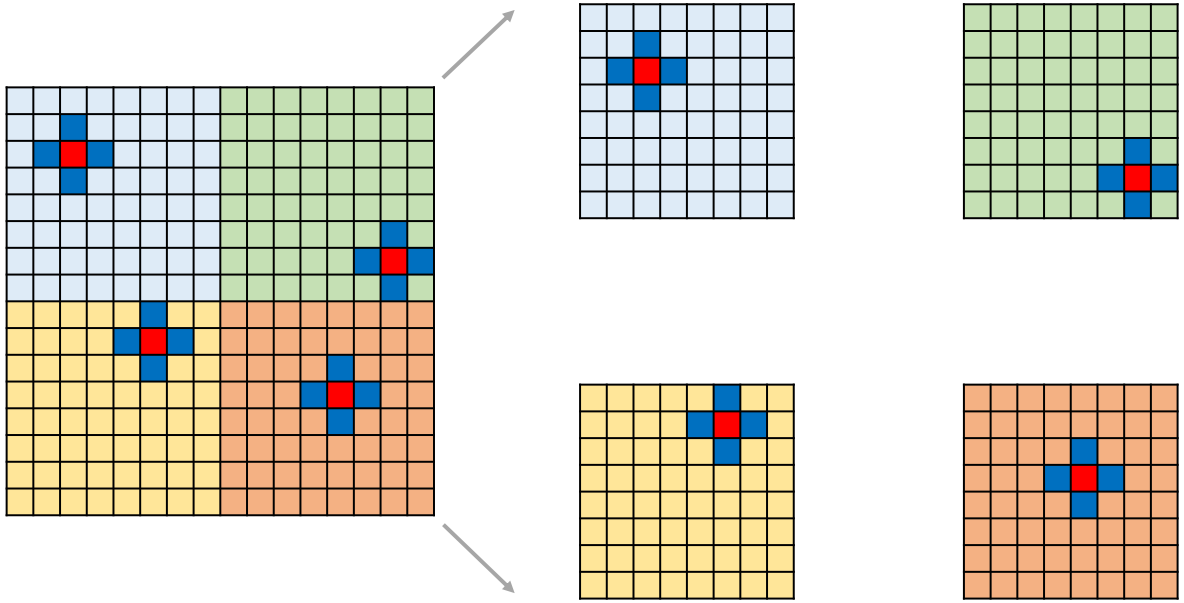
Communications overhead (PDE example)



Computational fluid dynamics (CFD), magnetohydrodynamics (MHD), climate, weather and many other simulations involve solving systems of partial differential equations on a grid that is distributed across processors to achieve parallelization.

16x16 grid divided into four 8x8 chunks that are distributed across processes.

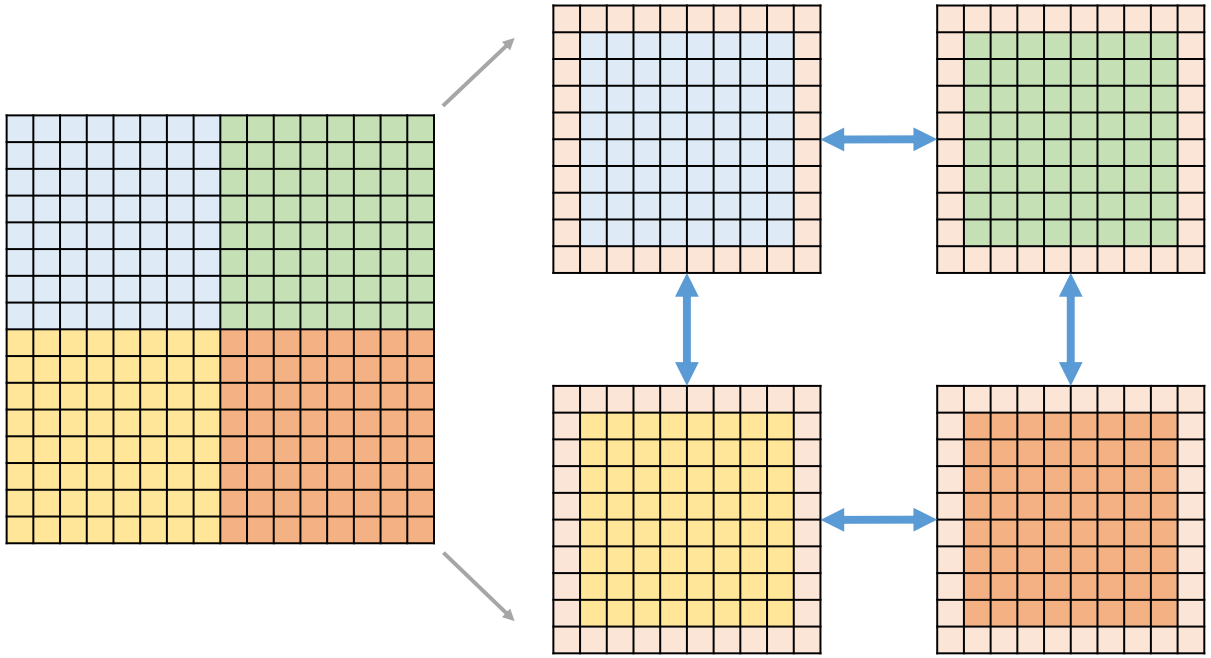
Communications overhead (PDE example)



Assume that each cell is updated using the values of the four neighboring cells.

For cells that are within the interior of each chunk, calculations can be done within each process.

Communications overhead (PDE example)

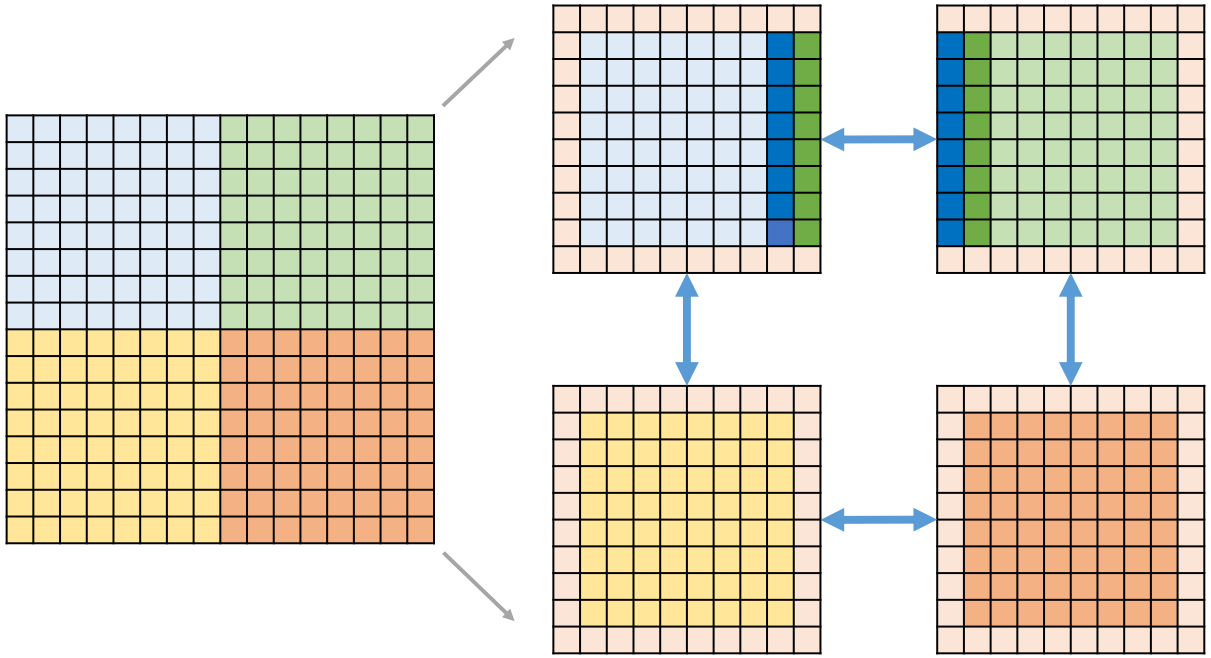


For cells at the boundaries of the chunks, data belonging to the neighboring processes is needed.

To accommodate this, we need a halo of ghost cells and data must be communicated between processes.

Data movement depends on both latency & bandwidth of the network and introduces communications overhead

Communications overhead (PDE example)

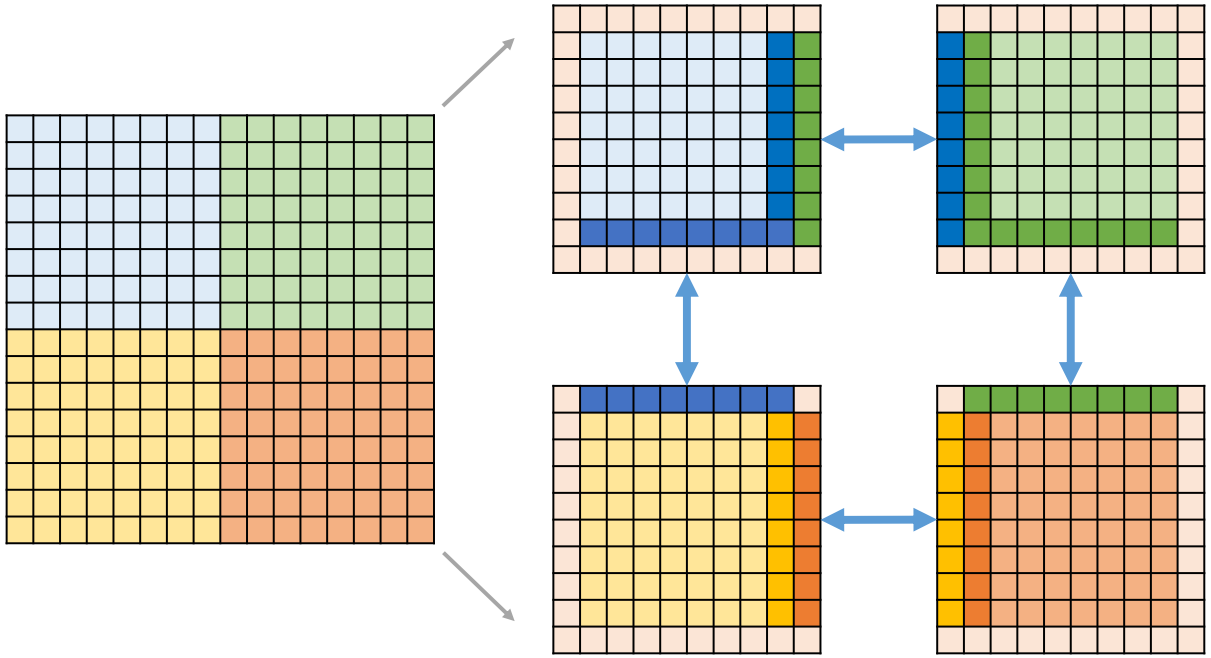


For cells at the boundaries of the chunks, data belonging to the neighboring processes is needed.

To accommodate this, we need a halo of host cells and data must be communicated between processes.

Data movement depends on both latency & bandwidth of the network and introduces communications overhead

Communications overhead (PDE example)



For cells at the boundaries of the chunks, data belonging to the neighboring processes is needed.

To accommodate this, we need a halo of host cells and data must be communicated between processes.

Data movement depends on both latency & bandwidth of the network and introduces communications overhead

All is not lost!

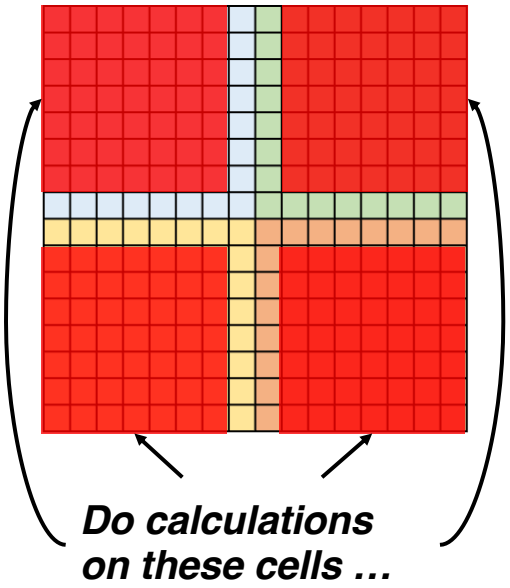
Between the hard limits imposed by Amdahl's law and all the other factors that affect scalability, how does anyone ever use all the cores on a single modern compute node, let alone the full power of large supercomputer?

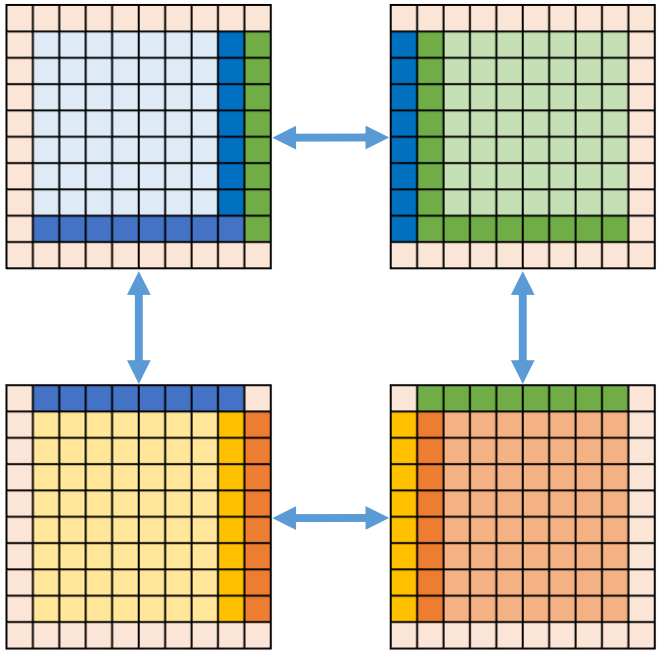
- The reality is that most parallel applications do not scale to thousands or even hundreds of cores
- The applications that achieve high scalability employ several strategies
  - Grow the problem size with the number of core or nodes (Gustafson's Law)
  - Overlap communications with computation
  - Use dynamic load balancing to assign work to cores as they become idle
  - Increase the ratio of computation to communications

Problem size limitations - solve a larger problem

32 “Hello Kitty” can easily paint a much larger fence in 1/32 of the time

Communications overhead – overlap with computation



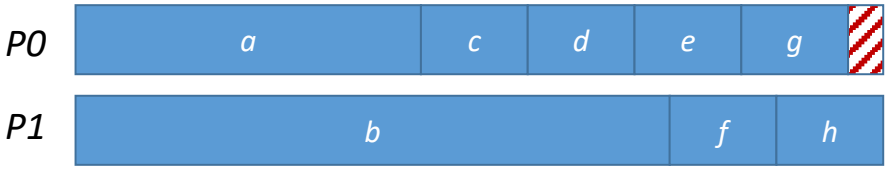


… while we wait for these communications to finish

Use dynamic load balancing

Imagine we have a set of tasks that could all be performed independently and that take varying amounts of time. Static assignment of tasks to processors can lead to large load imbalance. Can get better performance if we dynamically assign work to processors as they become idle.

First four tasks ("a-d") assigned to P0 and last four tasks ("e-h") assigned to P1



- *Task "a" assigned to P0 and task "b" to P1*
- *P0 finishes first and gets tasks "c", "d" and "e" before P1 finishes "b"*
- *P1 assigned "f" while P0 works on "e"*
- *P0 assigned "g" while P1 works on "f"*
- *P1 assigned "h"*

Table of contents

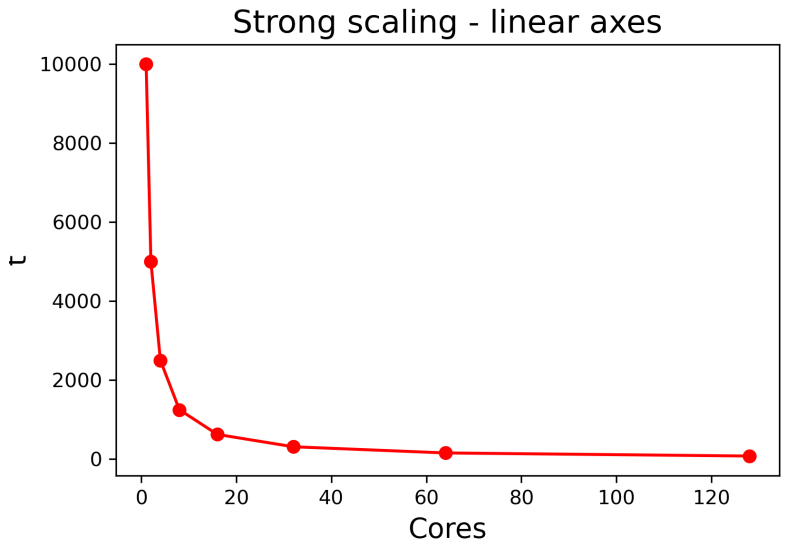
Running parallel applications

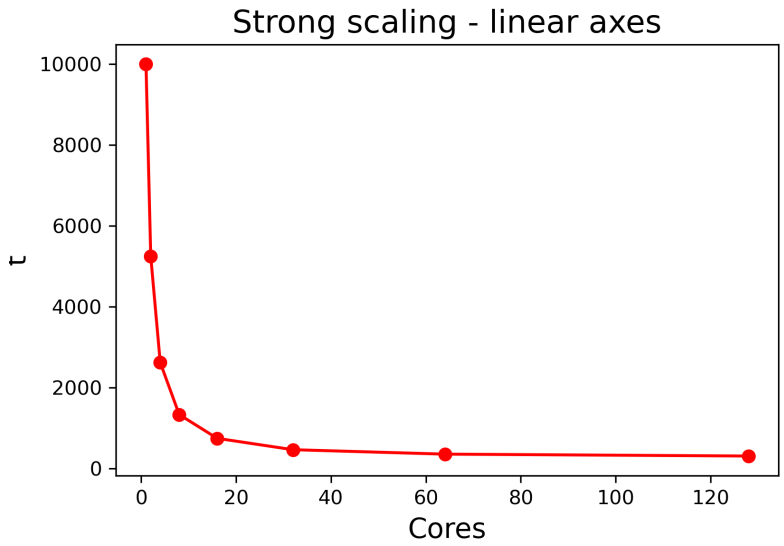
- So far, we've covered the basics concepts of parallel computing – hardware, threads, processes, hybrid applications, implementing parallelization (MPI and OpenMP), Amdahl's law and other factors that affect scalability.
- Theory and background are great, but how do we know how many CPUs/GPUs to use when running our parallel application?
- The only way to definitively answer this question is to perform a ***scaling study*** where a ***representative problem*** is run on different number of processors.

A representative problem is one with the same size (grid dimensions; number of particles, images, genomes, etc.) and complexity (e.g., level of theory, type of analysis, physics, etc.) as the research problems you want to solve.

Presenting scaling results (the wrong way)

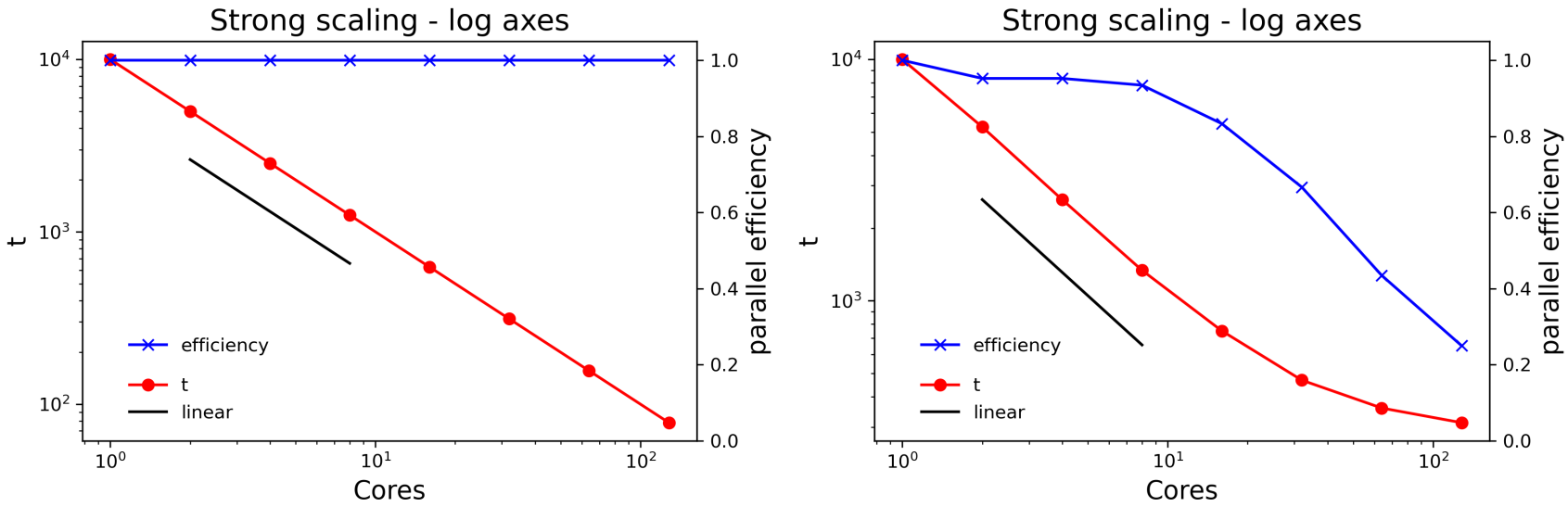
The plots on the left and right show the scaling curves for two different codes. Which one has the better scaling behavior? Kind of hard to tell since the timings at large core counts are indistinguishable.





Presenting scaling results (the right way)

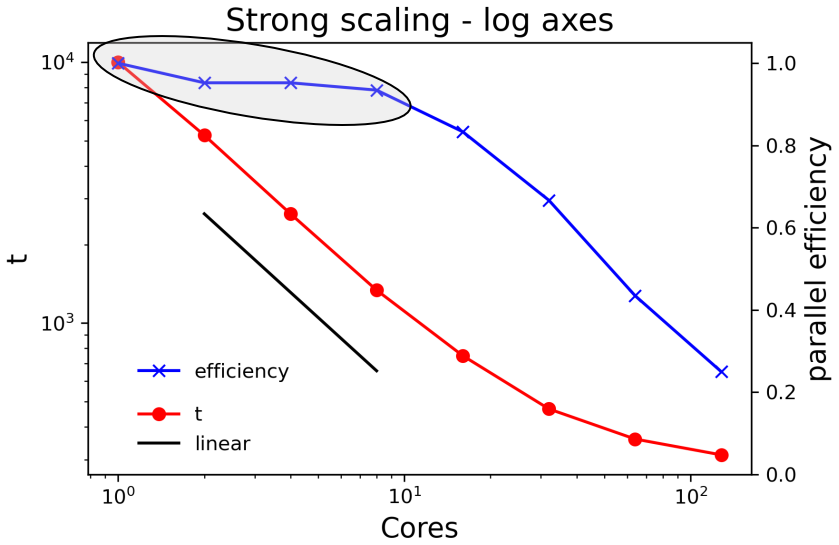
Plotting the same data on log axes gives a lot more insight. Note the different scales for the left axes on the two plots. Including a line showing linear scaling and plotting the parallel efficiency on the right axis adds even more value.



Where should I be on the scaling curve?

If your work is not particularly sensitive to the time to complete a single run, consider using a CPU/GPU count at or very close to 100% efficiency, even if that means running on a single core.

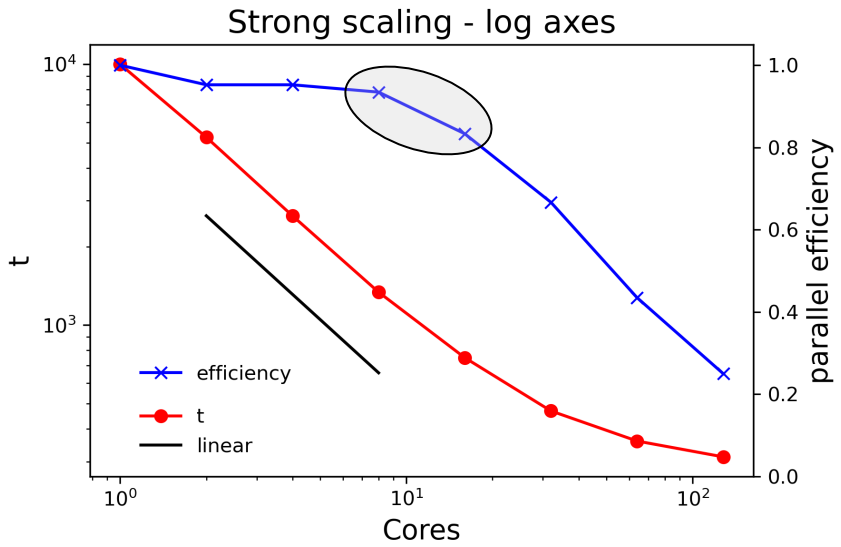
This especially makes sense for parameter sweep workloads where the same calculation is run many times with different sets of inputs.



Where should I be on the scaling curve?

Go a little further out on the scaling curve if the job would take an unreasonably long time at lower core counts or if a shorter time to solution helps you make progress in your research.

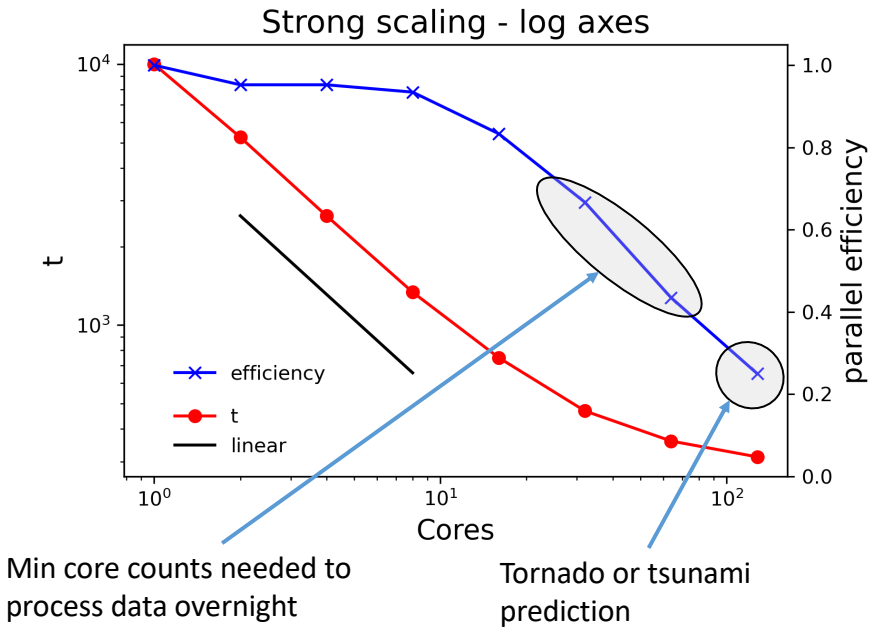
If code does not have checkpoint-restart capabilities and the run time would exceed queue limits, you'll have no choice but to run at higher core counts.



Where should I be on the scaling curve?

If the time to solution is absolutely critical, it's okay to run at lower efficiency.

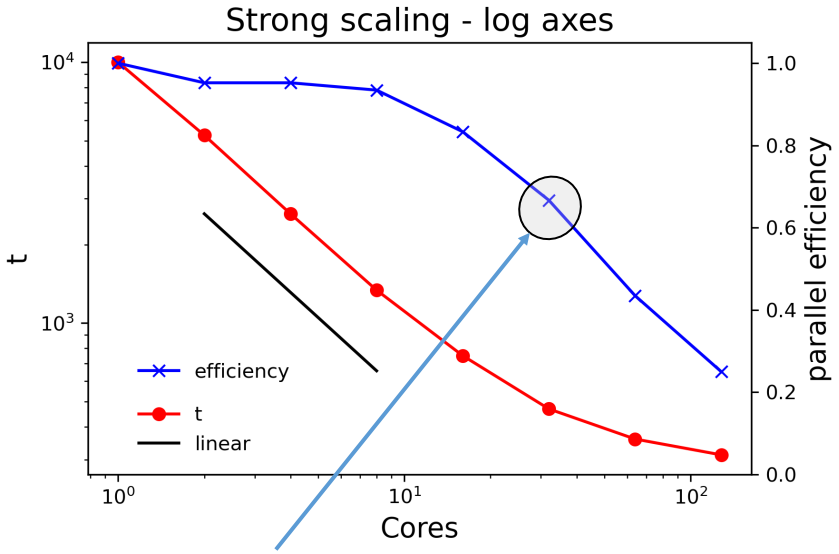
Examples might include calculations that need to run on a regular schedule (data collected during day must be processed overnight) or severe weather forecasting.



Other considerations – large memory footprint

Some ACCESS resources, such as *Expanse* (SDSC) and *Bridges-2* (PSC), allow multiple jobs to share a node. These jobs can normally request, without extra charge, memory in proportion to the core count. If memory is the limiting resource, it's okay to run at poorer scaling, but do consider using specialized large-memory nodes.

https://www.psc.edu/resources/bridges-2/user-guide-2/
https://www.sdsc.edu/support/user_guides/expanse.html



32 cores requested to get 64 GB of memory

Table of contents

We've only scratched the surface

- SDSC has many training resources covering a wide range of topics; we also expect that ACCESS will soon have a training catalog

  https://www.sdsc.edu/education_and_training/college_to_career.html

- User guides for nationally allocated resources contain practical information on job submission, accounting, compilation, data movement, available software and other site-specific content

  https://allocations.access-ci.org/resources
  https://www.sdsc.edu/support/user_guides/expanse.html

Conclusions

- Parallel computing is for everyone who wants to accomplish more research and solve more challenging problems
- You don't need to be a programmer, but you do need to know some of the fundamentals to effectively use parallel computers
- Processes are instances of programs; threads run within a process and access shared data; MPI and OpenMP are used to parallelize codes
- Amdahl's law imposes an *upper* limit of scalability, but there are other factors that impact scalability (load imbalance, communications overhead)
- Know how to display your scaling data and choose core counts

For code used to generate some figures in this presentation, see *https://github.com/sinkovit/Parallel-concepts*