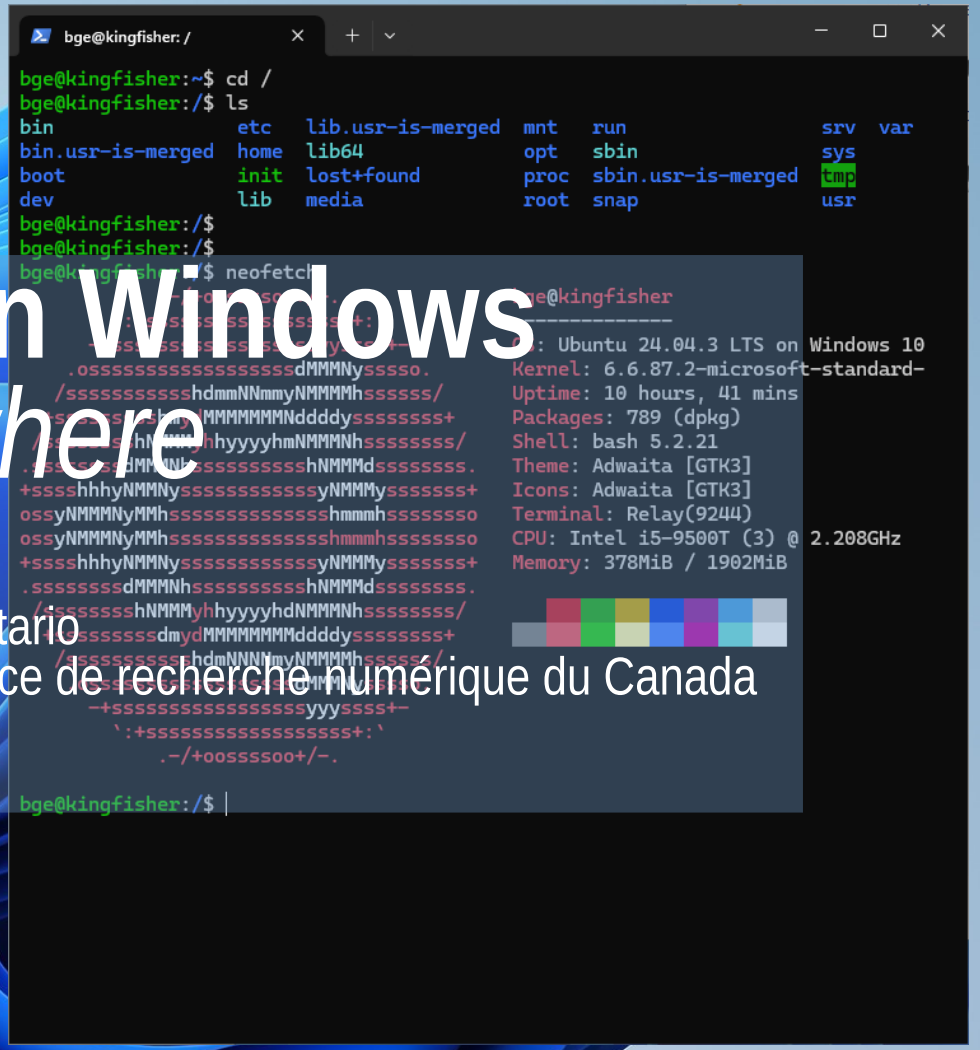
Running Linux on Windows
and taking it anywhere

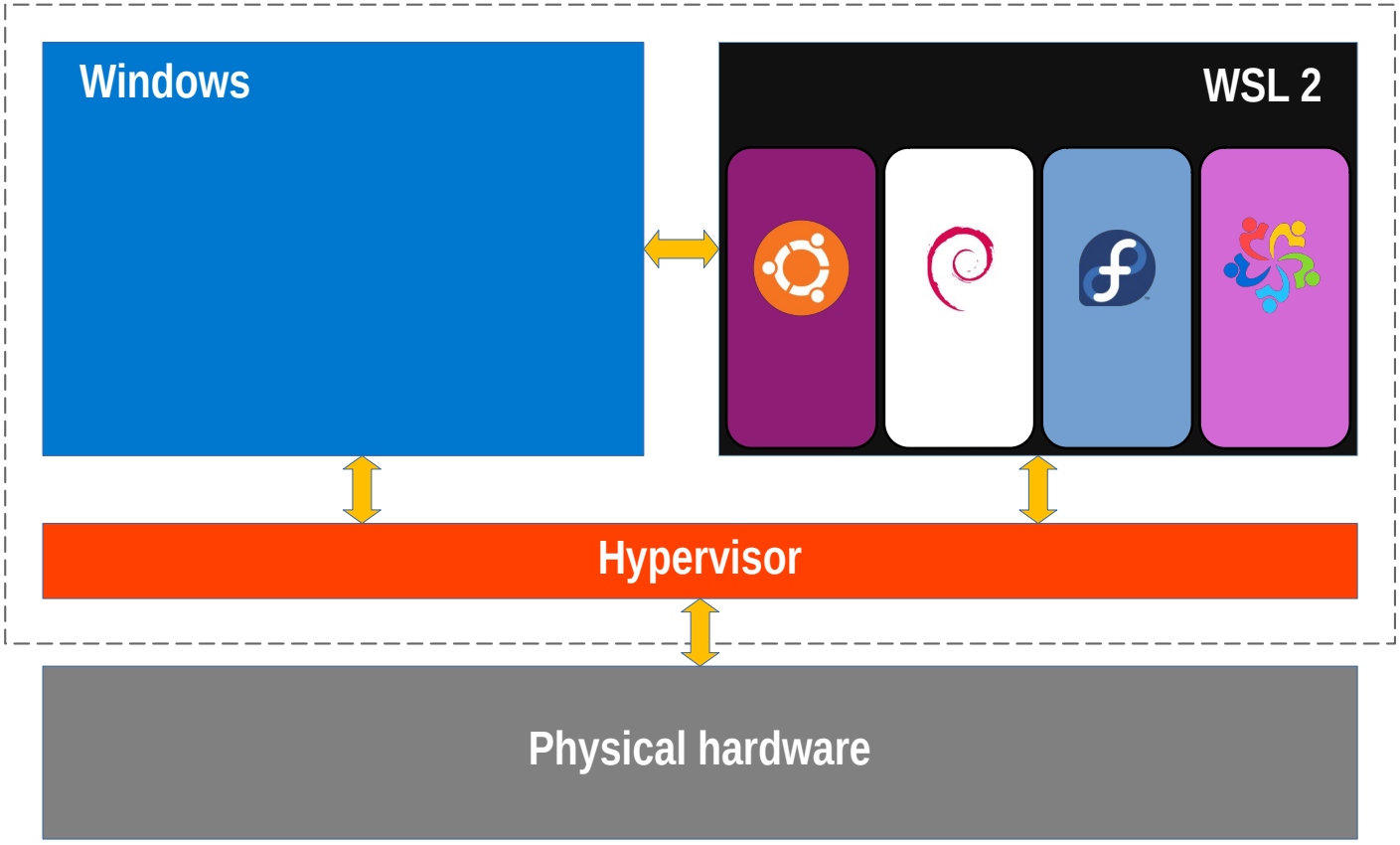
Ge Baolai, Western University
SHARCNET | Compute Ontario | Calcul Ontario
Digital Research Alliance of Canada | Alliance de recherche numérique du Canada



Outline

- What is and why Windows Subsystem for Linux (WSL)
- What do you need
- Best practices
- Developing multithreaded code in WSL
- Developing parallel MPI code in WSL
- Developing parallel Julia code in WSL
- Exporting your WSL Linux distro to clusters with Apptainer
- Reference

What is WSL



See also https://learn.microsoft.com/en-us/windows/win32/procthread/isolated-user-mode--ium--processes

Why WSL or the benefits

- One can do serious development work on Linux while still enjoying the features of Windows for day to day tasks.
- It is based on the virtualization technology, each Linux runs as a virtual machine, but with little overhead.
- One can develop, debug code in WSL on one's own computer before moving to clusters.
- Having it as a toolbox handy, one can work anywhere without Internet.
- One can run several different Linux distros at the same time, so allowing advanced users to explore more possibilities.

What do I need

- Windows 11.
- 8 GB RAM and 50GB storage for a Linux distro (e.g. Debian, Ubuntu, etc.)
- Any recent laptops, desktops with a minimum 16GB of RAM, enough disk space will meet the minimum requirements (check **Settings** → **System** → **About**).
- Internet connection to install the Linux.

What do I need

- Enable WSL on windows. Open a PowerShell as administrator, inside the terminal of PowerShell

  wsl --install

- To see a list of available Linux distro online

  wsl --list –online

- To install a Linux distro, say Debian

  wsl --install Debian

What do I need

Advanced settings configuration in WSL

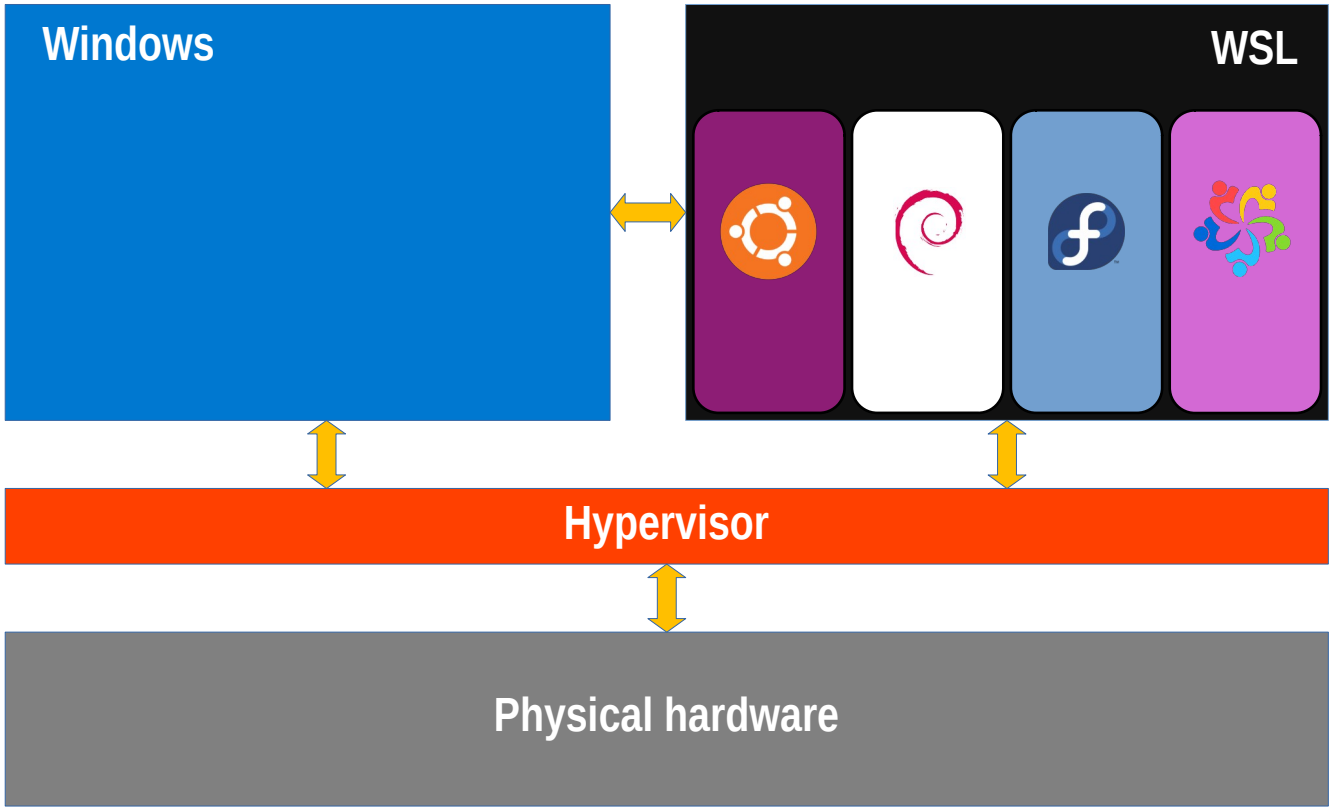
Per distro settings – /etc/**wsl.conf**

- Automount – what to be mounted when the system is launched.
- UID, GID, etc.
- Interop – wether allows WSL to launch Windows processes.
- GPU – wether allows WSL to access GPUs on the Windows host.
- Etc

Global setting – C:\Users\you**.wslconfig**

- Kernel – Microsoft built kernel, kernel modules, etc.
- Memory.
- Processors.
- Etc

What do I need



Notes and limitations:

- One can running flavours of Linux distros at the same time.
- Each one runs its own kernel.
- As of WSL 2, 2025, all Linux distros – VMs – running in WSL share the memory and cores set for WSL. So it is not possible to set the memory size or the number of cores per distro.
- All Linux VMs share the same network namespace. NAT is used.
- Bridged network is possible, but not easy.

What do I need

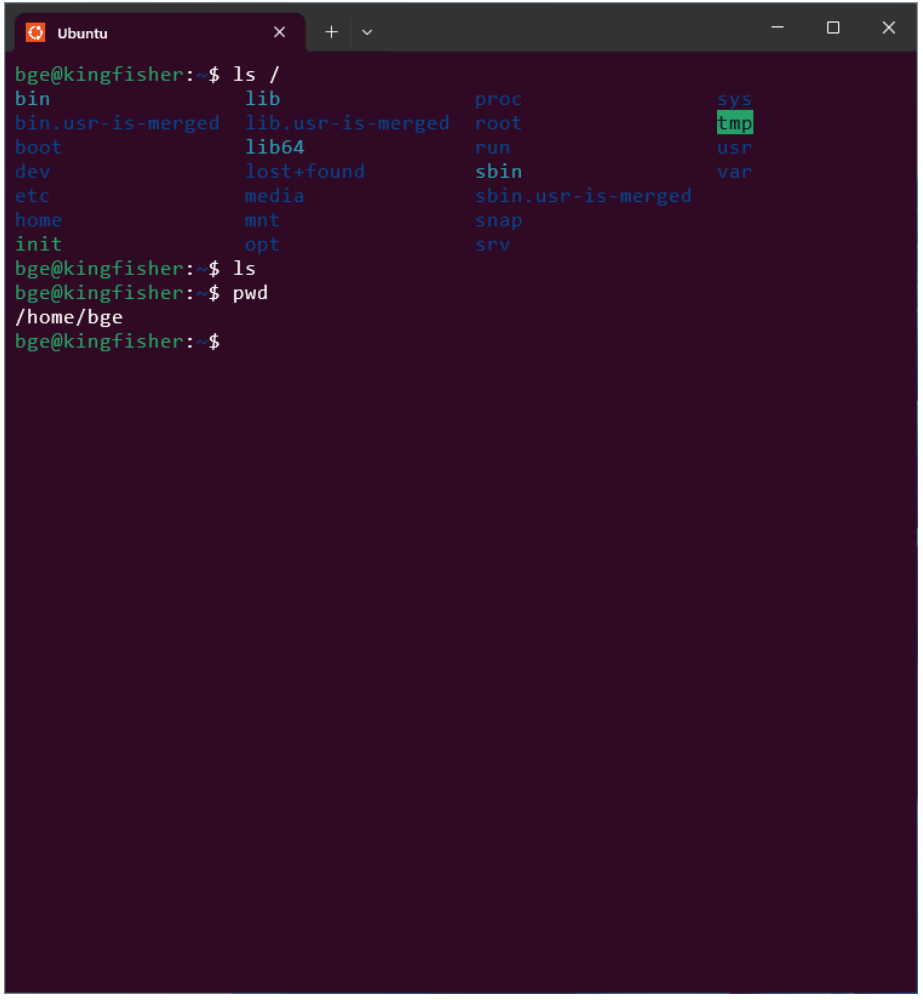
Sources

- How to install Linux on Windows with WSL, https://learn.microsoft.com/en-us/windows/wsl/install, Microsoft, 2025.
- Advanced settings configuration in WSL, https://learn.microsoft.com/en-us/windows/wsl/wsl-config, Microsoft, 2025.
- WSL repository on GitHub: https://github.com/Microsoft/wsl
- Ed Armstrong, Leveraging the power of Linux on Windows with WSL, https://www.youtube.com/watch?v=efHBV3b1Ylg, SHARCNET. 2023.

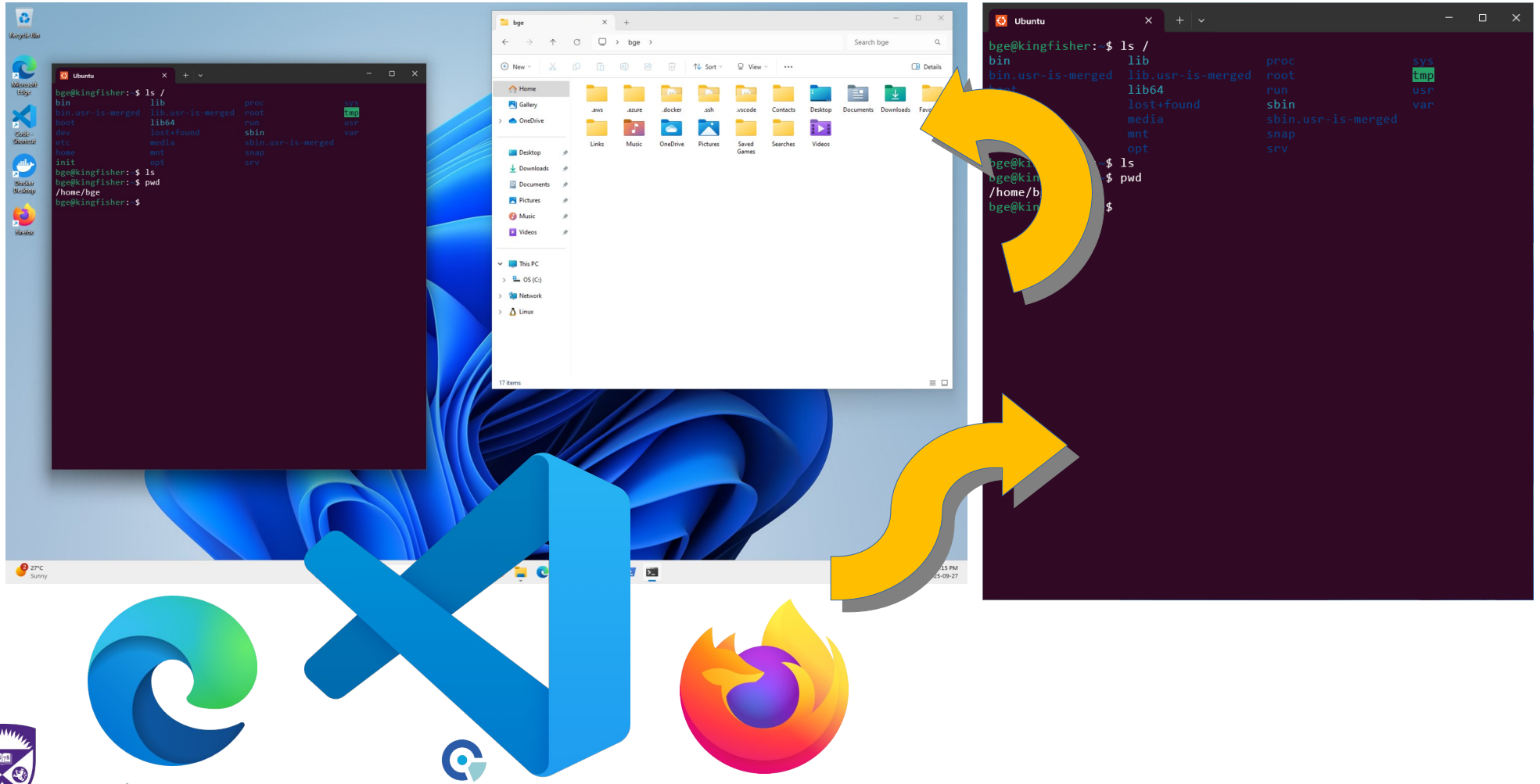
Best practices

Things to do

- Sharing file systems
- Development tools
  - Compilers
  - Debuggers
  - Languages
  - Libraries (BLAS, LAPACK, gsl, etc)
- Miscellaneous
  - curl, wget
  - ssh
  - nfs
  - ImageMagick
  - thunar
  - apptainer



Best practices



Best practices

Takeaways

- Windows and your Linux on WSL live together.
- You can access Windows files from WSL and vice versa.
- You can use VSCode to edit Windows files and files on WSL at the same time.
- You can do development work on Linux or through VSCode connected to WSL.
- For VSCode, you need to install **Remote Development** extension (which includes WSL).

Developing multithreaded code in WSL

Example: Summation of an array of 100,000,000 random numbers in parallel using threads

Purpose:

- Developing and testing a threaded code on personal computers.
- Examining the performance of threaded code.
- Being aware of the pitfalls of parallel processing.

Developing multithreaded code in WSL

Code snippet

```
#include <chrono>
#include <cstddef>
#include <execution>
#include <iomanip>
#include <iostream>
#include <limits>
#include <numeric>
#include <random>
#include <type_traits>
#include <vector>

constexpr std::size_t DATA_SIZE{ 100'000'000 };

using NUM_TYPE = double;
#define UNIF_DIST_TYPE std::uniform_real_distribution<NUM_TYPE>

constexpr NUM_TYPE udist_min(0);
constexpr NUM_TYPE udist_max(99);
```

```
using namespace std;

int main()
{
  vector<NUM_TYPE> v;
  v.reserve(DATA_SIZE);
  auto const t0 = chrono::steady_clock::now();
  generate_n(back_inserter(v), DATA_SIZE,
    [ ]( )
    {
      static random_device rd;
      static default_random_engine re{rd()};
      static UNIF_DIST_TYPE udist{ udist_min, udist_max};
      return udist(re);
    }
  );
  auto const t1 = chrono::steady_clock::now();
  auto const seqsum = accumulate(v.begin(), v.end(), NUM_TYPE{});
  auto const t2 = chrono::steady_clock::now();
  auto const parsum = reduce(execution::par_unseq, v.begin(), v.end(), NUM_TYPE{});
  auto const t3 = chrono::steady_clock::now();

  //cout << setprecision(numeric_limits<NUM_TYPE>::max_digits10);
  cout << "RNG generation time: " << chrono::duration<double>(t1-t0).count() << " (sec)\n";
  cout << "Sequential sum: " << setprecision(numeric_limits<NUM_TYPE>::max_digits10)
    << seqsum << ", time: " << chrono::duration<double>(t2-t1).count() << " (sec)\n";
  cout << "Parallel sum: " << setprecision(numeric_limits<NUM_TYPE>::max_digits10)
    << parsum << ", time: " << chrono::duration<double>(t3-t2).count() << " (sec)\n";
}
```

Developing multithreaded code in WSL

Takeaways

- You have access to muticores in WSL, as on clusters.
- Make sure your code works on your computer.
- WSL sees the same number of cores as the host Windows. You or may not see the ideal speedup due to multi factors
  - Your computer doesn't have many cores.
  - You hit false-sharing.
  - Your Windows is busy.
- Pay attention to the correctness.

Developing parallel MPI code in WSL

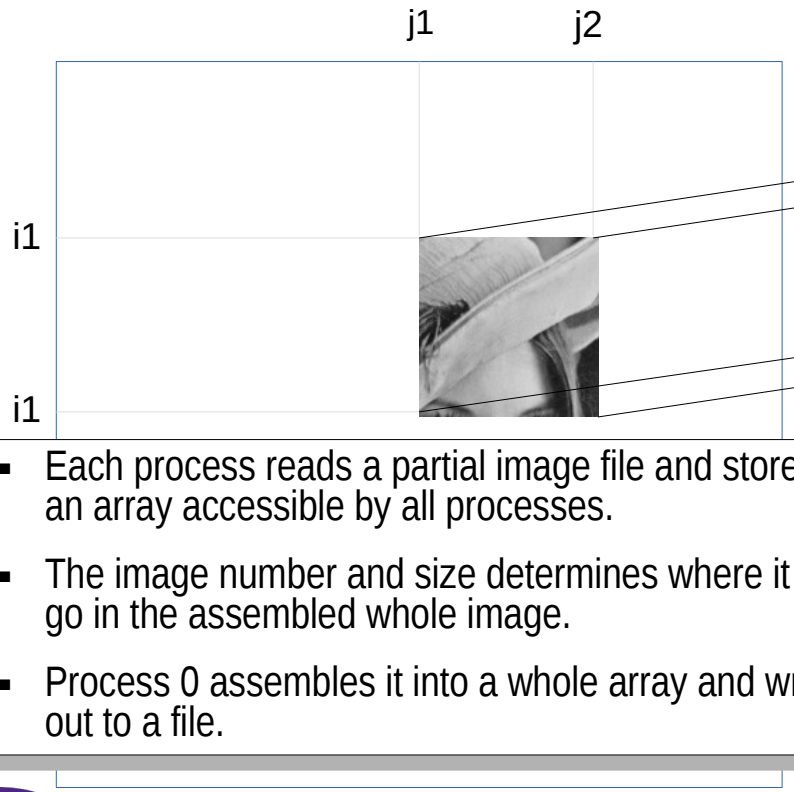
Example: Assemble partial images of Lenna into a whole using multiple processes. A Fortran code is shown to demonstrate parallel processing using message passing interface (MPI) library via the Fortran language feature. No explicit MPI calls are used.

Purpose:

- Developing and testing a parallel MPI code on a personal computer.
- Showcase the Fortran language feature that makes parallel programming high level and simple.

Source: https://en.wikipedia.org/wiki/Lenna

Developing parallel MPI code in WSL



- Each process reads a partial image file and stores in to an array accessible by all processes.
- The image number and size determines where it should go in the assembled whole image.
- Process 0 assembles it into a whole array and writes it out to a file.

Source: https://en.wikipedia.org/wiki/Lenna

Developing parallel MPI code in WSL

Code snippet

```
program lenna
  implicit none
  integer, allocatable:: pic(:,:), pic_p(:,:)[:]
  integer:: nx, ny, nx_p[*], ny_p[*], depth, p, q
  integer:: id, i, i1, i2, j1, j2, row, col
  character(80):: base_name[*], img, fmt

  ! Process 0 (image 1) reads parameters for restoring the image
  if (this_image() == 1) then
    open(10,file='input.dat',status='old')
    read(10,'(a)') base_name
    read(10,*) nx, ny
    read(10,*) p, q
    close(10)
    allocate(pic(nx,ny))
  endif
```

```
  ! Each process loads a piece of PGM image file into array pic_p
  sync all

  write(img,*) this_image()-1  ! Make string of image (proc) number
  img = adjustl(img)   ! Remove leading spaces
  img = trim(base_name[1])//'_'//trim(img)//'.pgm' ! e.g. Lenna_2.pgm

  open(10,file=img,status='old')
  read(10,*)   ! Skip the header of magic string "P2"
  read(10,*) nx_p, ny_p
  read(10,*) depth

  allocate(pic_p(nx_p,ny_p)[*])

  read(10,*) pic_p
  close(10)
  pic_p = transpose(pic_p)

  sync all
```

Developing parallel MPI code in WSL

cont'd

```
! Each process loads a piece of PGM image file into array pic_p
sync all

write(img,*) this_image()-1  ! Make string of image (proc) number
img = adjustl(img)   ! Remove leading spaces
img = trim(base_name[1])//'_'//trim(img)//'.pgm' ! e.g. Lenna_2.pgm

open(10,file=img,status='old')
read(10,*)   ! Skip the header of magic string "P2"
read(10,*) nx_p, ny_p
read(10,*) depth

allocate(pic_p(nx_p,ny_p)[*])

read(10,*) pic_p
close(10)
pic_p = transpose(pic_p)

sync all
```

```
! Process 0 collects pieces of image from others and restore them
if (this_image() == 1) then
  ! Collect pieces from other processes and assemble them
  pic(1:nx_p,1:ny_p) = pic_p
  do i = 2, num_images()
    row = (i-1) / q
    col = mod((i-1),q)
    i1 = row*ny_p + 1
    i2 = i1 + ny_p[i] - 1  ! Last one might be smaller
    j1 = col*nx_p + 1
    j2 = j1 + nx_p[i] - 1  ! Last one might be smaller
    pic(i1:i2,j1:j2) = pic_p(:,:)[i] ! Copy data from process i
  enddo

  ! Write the assembled image to a PGM file
  img = trim(base_name)//'_whole'//'.'//'pgm'      ! File name Lenna.pgm
  open(11,file=img,status='unknown')
  write(11,'("P2")')
  write(11,'(i3,i4)') nx, ny
  write(11,'(i3)') depth
  write(fmt,'(i3)') nx-1
  fmt = '(i3,'//trim(fmt)//'i4)'        ! Output format, e.g. '(i3,255i4)'
  do i = 1, ny
    write(11,fmt) pic(i,:)
  enddo
  close(11)
endif
end program lenna
```

Developing parallel MPI code in WSL

Takeaways

- You can run MPI code with multiple processes (“ranks”) in WSL.
- It is very easy to run parallel code in Fortran without knowing MPI.
- Your personal computer normal doesn’t have many cores. You can fake it by using a machine file to define the “hosts” on which you want to run the MPI code. In this case, the hosts should be all **localhost**.
- If using OpenMPI, the host file contains

  localhost slots=N

  where N is number of cores, you may set it arbitrarily. With OpenMPI 5, you would need option –bind-to :overload-allowed, eg.

  mpirun -n 8 --hostfile hostfile --bind-to :overload-allowed ./my_mpi_prog

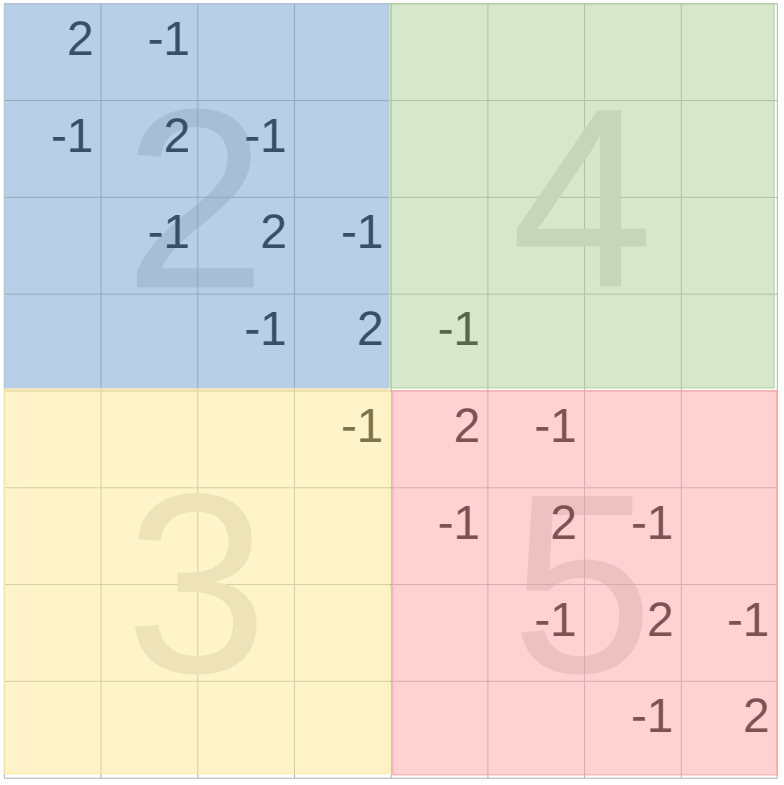
  cafrun -n 8 --hostfile hostfile --bind-to :overload-allowed ./my_mpi_prog

Developing parallel Julia code in WSL: Distributed arrays

Example: A matrix stored across 4 processes on a 2x2 Cartesian processor grid

Proces 1 has the blue portion.

But it also has access to other portions stored remotely, *simply via indices.*

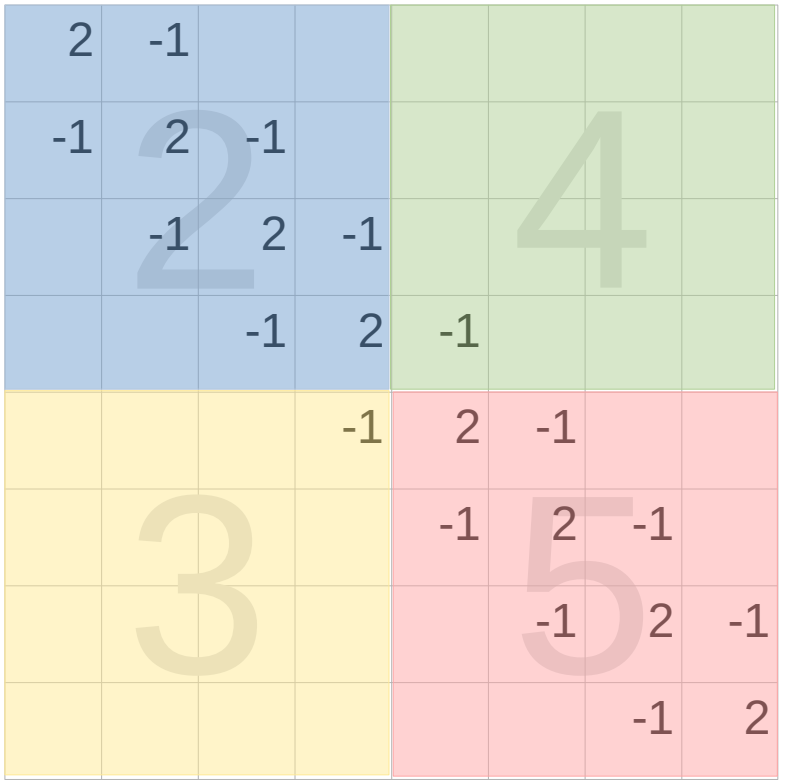


Suitable for handling large data sets that can NOT fit on a single machine.

Developing parallel Julia code in WSL: Distributed arrays


```
using Distributed, DistributedArrays
@everywhere using LinearAlgebra
@everywhere function bod(n) # Blocks on diagonal
    a = zeros(n,n)
    a[diagind(a,0)] .= 2.0
    a[diagind(a,-1)] .= -1.0
    a[diagind(a,1)] .= -1.0
    return a
end
@everywhere function lob(n) # Lower off-diagonal blocks
    a = zeros(n,n); a[1,n] = -1.0;
    return la
end
@everywhere function uob(n) # Upper off-diagonal blocks
    a = zeros(n,n); a[n,1] = -1.0;
    return a
end
```

Matrix A distributed on 4 processors on a 2x2 grid



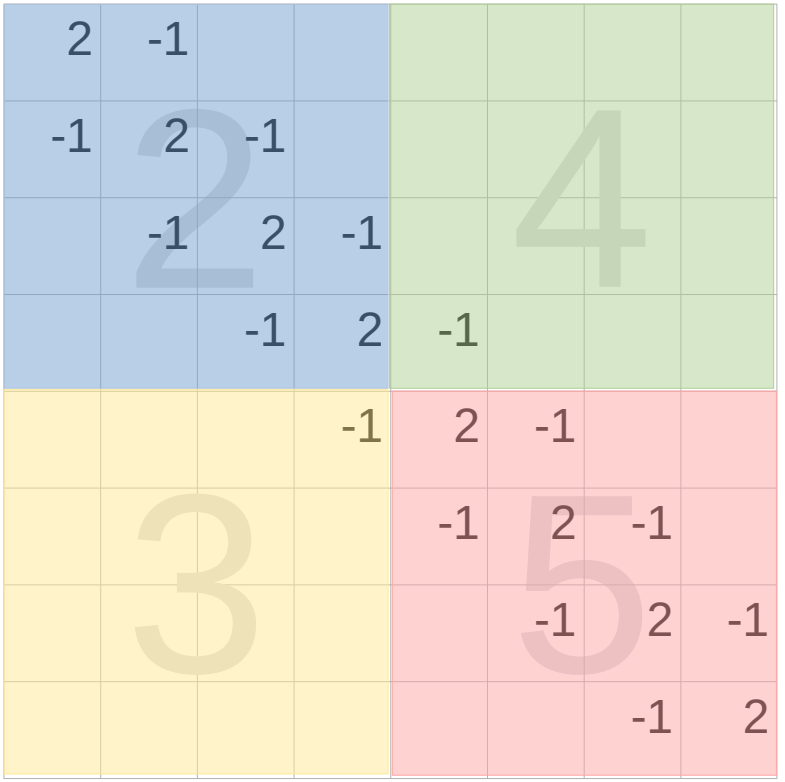
Developing parallel Julia code in WSL: Distributed arrays


```
# Call functions on workers to created local portions
d11 = @spawnat 2 bod(4)
d12 = @spawnat 3 uob(4)
d21 = @spawnat 4 lob(4)
d22 = @spawnat 5 bod(4)

# Create a distributed matrix on a 2x2 processor grid
DA = DArray(reshape([d11 d21 d12 d22],(2,2)));

# Access components owned by worker 3
display(DB[5:8,1:4])
```

4×4 view(::DArray{Float64,2,Array{Float64,2}}, 5:8, 1:4) with eltype Float64:

```
 0.0  0.0  1.0  -4.0
 0.0  0.0  0.0   1.0
 0.0  0.0  0.0   0.0
 0.0  0.0  0.0   0.0
```

Matrix A distributed on 4 processors on a 2x2 grid



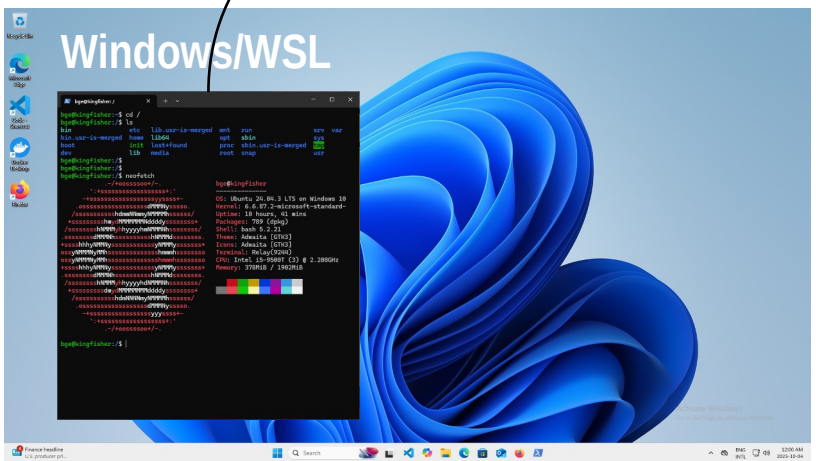
Developing parallel Julia code in WSL: Distributed arrays

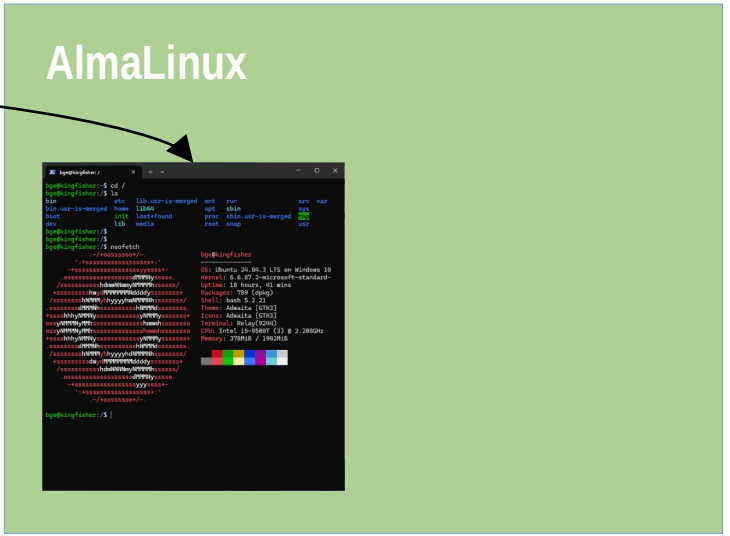
Takeaways

- Julia provides a new potential for writing large scale parallel code.
- It's high level, easier to write parallel code compared to MPI code.
- With the support for distributed and shared arrays, writing parallel code at a high level becomes much easier.

Exporting your WSL Linux distro to clusters with Apptainer

Run as a VM





Run in Apptainer

Exporting your WSL Linux distro to clusters with Apptainer

A receipe for making an apptainer image from a live WSL Linux distro:

- You need to install **Docker Desktop** on Windows. The start the Docker Desktop.
- Then export the current Linux distro from WSL as a tarball

  wsl --export Ubuntu C:\Users\bge\Documents\wsl-backups\wsl-ubuntu.tar
- Convert the tar ball to Apptainer ready input using docker

  cd C:\Users\bge\Documents\wsl-backups

  docker import wsl-ubuntu.tar wsl-ubuntu:20251005

  docker save wsl-ubuntu:20251005 -o wsl-ubuntu-docker.tar
- In a WSL Linux distro, create an Apptainer image from the docker converted tarball

  sudo **apptainer** build wsl-ubuntu.sif docker-archive:wsl-ubuntu-docker.tar
- Run the apptainer on a Linux system

  apptainer shell wsl-ubuntu.sif

Reference

1. How to install Linux on Windows with WSL, https://learn.microsoft.com/en-us/windows/wsl/install, Microsoft, 2025
2. Ed Armstrong, Leveraging the power of Linux on Windows with WSL, https://www.youtube.com/watch?v=efHBV3b1Ylg, SHARCNET. 2023.
3. Best practices for setting up a WSL development environment, https://learn.microsoft.com/en-us/windows/wsl/setup/environment, Microsoft.
4. C++ Reference, https://www.cppreference.com/
5. Michael Metcalf, John Reid, Malcolm Cohen, Modern Fortran Explained, Oxford University Press, 2014.
6. Julia Documentation, https://docs.julialang.org/en/v1/