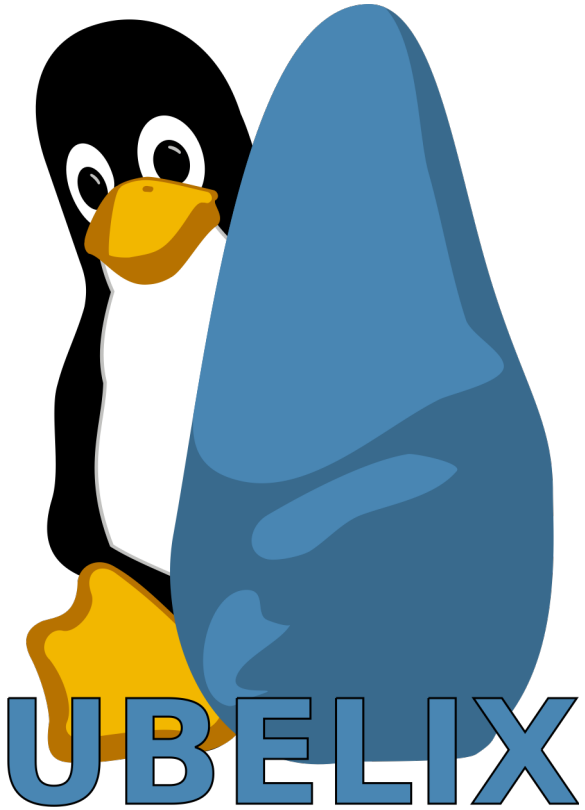
UNIVERSITÄT BERN

Advanced Usage of UBELIX

Science IT Support (ScITS)

Michael Rolli, Nico Färber, IT-Services Office

Contact: grid-support@id.unibe.ch



Agenda

- Checkpointing/Restart
- GNU Parallel
- Job Steps (srun)
- Interactive Jobs (salloc/srun)
- Run GUI Applications on UBELIX
- Parallel Computing
- GPUs @ UBELIX

Who we are

The sysadmins

Michael Rolli, MD

- Master Humanmedizin 2001 at UniBE
- 2001 – 2013: Institute of Medical Education (IML)
- Since 2013: Full-time sysadmin UBELIX

Nico Färber

- Degree in Computer Science
- 2008 – 2016: Part-time sysadmin ID
- Since 2016: Full-time sysadmin UBELIX

Before we Dive in…

Have you activated your CA for UBELIX?

- Campus Account must be activated for UBELIX
- Subscribe to our mailing list:
  - https://listserv.unibe.ch/mailman/listinfo/grid-users/
  - 2-10 mails per month
  - Important announcements (maintenance downtime,…)

Keep in mind…

Some guidelines

- Know your software/job
  - Know how to tweak it
  - Know how it uses resources, e.g. is it greedy?
  - Know its resource demands, e.g. memory consumption
  - Know its dependencies, e.g. additional libraries
  - Know its limitations, e.g. does it run in parallel?
- Be a good citizen
  - https://docs.id.unibe.ch/ubelix/code-of-conduct

Hands-on

Copy the examples

- Copy the example folder to your home directory
    - cp -r /storage/software/workshop-adv/ $HOME

Checkpointing/Restart

Save program state

- Save state information/intermediate results of a job/computation
- Restart job/computation from a previously saved state

Checkpointing/Restart

Why checkpointing

- Do not loose all work upon node failure or unexpected job termination
- Run a time consuming job in a partition that only allows short runtimes
- Run a job in a hostile environment (e.g. GPU partition)
- Checkpointing is **mandatory** when using the GPU partition

Checkpointing/Restart

Levels of checkpointing

- External program to checkpoint your job (not supported yet)
  - BLCR, …
  - Suitable for proprietary (closed source) software
- Embed checkpointing logic within your code
  - Must have access to source code
  - Not transparent to developer, portable
- Some applications come with built-in checkpointing capability:
  - Gaussian, Quantum Espresso, CP2K, …

Checkpointing/Restart

General recipe

- Check if there is a previously saved state
- If yes, restart from saved state
- If no, bootstrap
- Periodically save the state of a running job (e.g. triggered by external signal)

Checkpointing/Restart

How can Slurm help?

- Slurm sends signal to job 60s before termination
  - *SIGTERM* followed by *SIGKILL* after grace period
- User can explicitly send signal to job using *scancel*
  - *scancel --signal=USR1 <jobid>*
- Catch (trap) signals in code and act accordingly (Bash, Python, C/C++, …)

Checkpointing/Restart

C/C++

```c
#include <signal.h>  // C
#include <csignal>   // C++

void signal_handler(int signal) {
  // Save program state and exit
  (...)
 exit(0);
}

// Register signal handler for SIGTERM
signal(SIGTERM, signal_handler);  // signal_handler: function to handle signal
(...)
```

Checkpointing/Restart

Python

```python
#! /usr/bin/env python
import signal
import sys


def signal_handler(sig, frame):
  # Save program state and exit
  (...)
  sys.exit(0)


signal.signal(signal.SIGTERM, signal_handler)
(...)
```

Checkpointing/Restart

A simple example

- Run job checkpointing/job.sh
    - Catch signal *SIGUSR1* and *SIGTERM*
- Use *srun* to run the binary for proper signal handling
- Verify that the current state was written to *state.log*
- Kill job: ***scancel <jobid>***
- Start job again. Verify that job continues from last known state

GNU Parallel

https://www.gnu.org/software/parallel

- Execute shell scripts in parallel (parallelization based on input data)
- Allows to restart/continue from last task executed
- SLURM: Can be used to distribute a set of tasks among a number of workers
- Particularly useful when number of tasks >> number of workers

GNU Parallel

Example (1/2)

- Install GNU Parallel: Run gnu_parallel/install.sh
- Run example gnu_parallel/exercise01/compress.sh

```
Start compressing files sequentially.
22.51user 1.45system 0:24.28elapsed 98%CPU (0avgtext+0avgdata 912maxresident)k
0inputs+0outputs (0major+713minor)pagefaults 0swaps
Done.
Start compressing files in parallel on 2 CPUs.
24.28user 1.63system 0:16.52elapsed 156%CPU (0avgtext+0avgdata 14684maxresident)k
0inputs+2808outputs (0major+18689minor)pagefaults 0swaps
Done.
Start compressing files in parallel on 4 CPUs.
24.52user 1.69system 0:06.98elapsed 375%CPU (0avgtext+0avgdata 14692maxresident)k
0inputs+2400outputs (0major+18873minor)pagefaults 0swaps
Done.
```

- Nice, but what about checkpointing/restart?

GNU Parallel

Example (2/2)

- Run example gnu_parallel/exercise02/job.sh
- Important options: ***--joblog*** and ***--resume/--resume-failed***
- Cancel the job after about 30s
- Restart job while inspecting logs/runtasks.log

srun

Create job steps

- If *srun* called outside an existing allocation (salloc or sbatch)
  - Implicit allocation of resources
- If *srun* called within an existing allocation (salloc or sbatch)
  - Use all/subset of the resources of the allocation
- With array jobs, each array task has its own allocation
- With srun we can start multiple tasks within the same allocation

srun

Run job steps concurrently

- Run srun in the background to run job steps concurrently
  - *srun -N1 -n1 --exclusively … &*

- Wait for background tasks to finish before exiting job script
  - *srun -N1 -n1 --exclusively … &*
  - *srun -N1 -n1 --exclusively … &*
  - *(…)*
  - *wait*

srun

Example – Run serial tasks concurrently

- Submit job *res_management/job.sh*
- Show information about job steps
  - *sacct -j <jobid> --format=jobid,start,elapsed,ncpus,node,state,exitcode*
- Hint: *sacct --helpformat* for a list of all format options

Example – Run parallel tasks

- Open MPI has Slurm support build in and vice versa
  - *srun --mpi=pmi2 my_mpi_app*
- „Open MPI automatically obtains the list of hosts and how many processes to start on each host from Slurm directly“
  - No need to specify *--nodelist*, *--host* or *--np* options to *mpirun*
- *$SLURM_NTASKS* corresponds to number of MPI ranks
- Submit job parallel_tasks/job.sh
- Submit job parallel_tasks/job_v2.sh and show job steps

Interactive Jobs

Work interactively

- Create an allocation using salloc
  - salloc --nodes=1 --ntasks-per-node=4 --time=00:30:00
  - Blocks until resources are available
- Use srun to create job steps
- Good for iterative testing/debugging!
- *srun [options] --pty bash*
  - interactive shell on first compute node of the allocation

Run GUI Apps on UBELIX

X11 forwarding

- X-Server on your local machine!
  - Mac (X11 no longer included): Xquartz
  - Windows: Xming
- Enable X11 forwarding from the login node to your local machine:
  - *ssh -Y -l <username> submit.unibe.ch*
- Public keypair to communicate password-less between the nodes
- (srun|salloc) [options] --x11 --pty bash
  - --x11: Sets up X11 forwarding on all allocated nodes
  - --pty: Pseudo terminal that runs the command (srun only)

Run GUI Apps on UBELIX

Example - Running Matlab

```
salloc -N1 -n1 --mem-per-cpu=4G --time=00:30:00 --x11
module load MATLAB
srun --pty matlab
```

- Alternatively:

```
srun -N1 -n1 --mem-per-cpu=4G --time=00:30:00 --x11 --pty bash
module load MATLAB
matlab
```

- error: No DISPLAY variable set, cannot setup x11 forwarding. Did you login with *ssh -Y …*?

Parallel Computing

Why parallel programming?

- No more free speedup!
- Many CPU cores available on modern computing hardware
- Your code may run faster if using multiple CPU cores
    - Whether this is true depends on the problem you try to solve
- Your application needs more memory than a single node provides

Parallel Computing

Three takeaways from the first course

- Be a good citizen, resources are scarce
  - This is even more important for parallel jobs!
- Know your software
  - E.g. is your software capable of leveraging parallelism
- No generic way to covert a sequential program to a parallel program

Parallel Computing

Does your job run in parallel? (1/2)

- If you don’t know for sure, verify it!
- Use *scontrol* to show CPU IDs allocated to your job
  - *scontrol **–d** show job <jobid>*

```
TRES=cpu=16,mem=32G,node=1,billing=16
Socks/Node=* NtasksPerN:B:S:C=16:0:*:* CoreSpec=*
  Nodes=anode041 CPU_IDs=0-15 Mem=32768 GRES_IDX=
MinCPUsNode=16 MinMemoryCPU=2G MinTmpDiskNode=0
Features=(null) DelayBoot=00:00:00
```

- Now you know the IDs of the allocated CPU cores. What next?

Parallel Computing

Does your job run in parallel? (2/2)

- Verify that the processes use the allocated CPU cores
- On the allocated compute node(s):
  - *top -H -u $USER* and activate field „Last Used CPU“
  - *ps -T c -u $USER -o pid:10,ppid:10,spid:10,rss:10,psr:6,state:6,time:10,cmd*
  - or use convenience script (see next example)

Parallel Computing

Example - Gathering diagnostics

- Submit job *diag/job.sh*
- Run fdiag.sh as a job step under an already allocated job
  - *srun --jobid=<jobid> fdiag.sh*

Parallel Computing

What about CPU efficancy?

- Does your job use the allocated CPUs efficiently?
  - ./cpu_efficancy/seff <jobid>
  - Contribution to Slurm by Princeton University
  - Also reports memory efficancy
    - Does not work with current version of Slurm!
  - Outlook: Embed information in end mail

Parallel Computing

What about scalability?

- Inherently serial part of a program
- Parallel overhead, i.e. communication overhead
- At what point does adding more cores no longer increase execution speed?
  - Check the manual of your software
  - Empirical tests: Allocate 2, 4, 8, ... CPU cores
- Does the speedup justify the additional CPU resources?

Parallel Computing

Shared memory computing

- Communication between processes is implicit and transparent
- Processes share the same memory
- Job limited to resources provided by a single compute node
- Implementations: OpenMP, ...

Parallel Computing

Dsitributed memory computing

- Communication between processes is explicit
  - Processes communicate by passing messages (MPI)
- Job can use resources from different compute nodes
- Communication overhead, minimize number of nodes!
- Use *--constraint=<feature>* to request a homogeneous set of nodes
- Implementations: Open MPI, ...

Parallel Computing

Request resources

- For shared memory jobs use
  - *--mem-per-cpu=2G*
  - ***--cpus-per-task=16***
  - SLURM will allocate 16 CPUs and 32G RAM on same node
- For distributed jobs use
  - *--mem-per-cpu=2G*
  - ***--nodes=4 --tasks-per-node=20***
  - SLURM will allocate 80 CPUs and 160G of RAM (4 nodes)

Parallel Computing

OpenMP

- OpenMP: API for writing multithreaded applications
- De-facto standard API for writing shared memory parallel applications
- Switches for compiling/linking:
  - gcc: -fopenmp
  - pgi: -mp
  - intel: /Qopenmpi
- Master thread spawns additional threads as needed

Parallel Computing

OpenMP (1/3)

- OpenMP: API for writing multithreaded applications
- De-facto standard API for writing shared memory parallel applications
- Switches for compiling/linking:
  - gcc: -fopenmp
  - pgi: -mp
  - intel: /Qopenmpi
- Master thread spawns additional threads as needed

Parallel Computing

OpenMP (2/3)

- Request a certain number of threads:
  - Set an initial value: export *OMP_NUM_THREADS=x*
  - *omp_set_num_threads(x)*
  - *num_threads(x)* clause
  - In either case use *$SLURM_CPUS_PER_TASK*
- Create threads with the parallel construct:
  - **#pragma omp parallel**
  - Each thread executes a copy of the code within the block

Parallel Computing

OpenMP (3/3)

```c
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

void main() {
        int a = 10;
        // Get number of cores allocated by SLURM
        const char* nc = getenv("SLURM_CPUS_PER_TASK");
        printf("Number of CPU cores allocated by SLURM: %s\n", nc);
        // Create thread pool
        omp_set_num_threads(6); << WRONG! DOES NOT MATCH ALLOCATED CPUS
        #pragma omp parallel
        // A copy of the following block is executed by each thread
        {
                // Integer 'a' is shared between all threads
                // thread_ID is different for each thread
                int thread_ID = omp_get_thread_num();
                printf("a equals %d and I am thread %d\n", a, thread_ID);
        }
}
```

```
#!/bin/bash

#SBATCH --mail-type=none
#SBATCH --cpus-per-task=8

# You code below this line
./omp_example
```

```
Number of CPU cores allocated by SLURM: 8
a equals 10 and I am thread 0
a equals 10 and I am thread 2
a equals 10 and I am thread 5
a equals 10 and I am thread 1
a equals 10 and I am thread 4
a equals 10 and I am thread 3
```

Parallel Computing

Open MPI

- Resource allocation dictates where the job will run
- --nodes=4 --tasks-per-node=20
  - 4 nodes, 20 MPI processes per node
- Open MPI build with Slurm support
  - mpirun ./mympi VS. mpirun -np $SLURM_NTASKS ./mympi
- Explicit inter-process communication
  - **MPI_SEND, MPI_RECEIVE**

Parallel Computing

Example - OMPI communication

- Compile source file *ompi/mpi_send_recv.c* using *mpicc*
  - *module load OpenMPI/3.1.1-GCC-7.3.0-2.30*
  - *mpicc -o mpi_send_recv mpi_send_recv.c*

- Submit job script *ompi/job.sh*
  - Load correct version of Open MPI

- Are the requested resource request appropriate for this job?

Parallel Computing

Matlab

- Built-in multithreading
  - Functions automatically execute on multiple computational threads (operations on arrays/matrices, ...)
  - *maxNumCompThreads(getenv($SLURM_CPUS_PER_TASK))*
- Explicit multiprocessing
  - Parallel Computing Toolbox (PCT)
  - Distributed Computing Server (DCS). Not available on UBELIX!

Parallel Computing

Python

- Shared memory
  - *multiprocessing* library
- Distributed memory
  - Python supports MPI through the *mpi4py* module
- GPU
  - Python supports Nvidia CUDA. See *pycuda* module

GPUs @ UBELIX

- UBELIX provides (09/2019):
  - 80 GeForce, fast single precision
  - 16 Tesla P100, fast double precision, no video output
- CPU code does not magically run on the GPU
  - You have to explicitly adapt your code to run on the GPU
- Code that runs on a GPU will not necessarily run faster than it runs on the CPU
  - GPUs are suitable for tasks that are **highly parallelizable**

GPUs @ UBELIX

Request GPU resources

- Must request a GPU partition
  - --partition=gpu
- Must request GPU cards
  - --gres=gpu:teslaP100:<number_of_gpus>
  - --gres=gpu:gtx1080ti:<number_of_gpus>
  - --constraint=gtx1080 or --constraint=rtx2080

GPUs @ UBELIX

Example – Hello, World! (1/2)

- CUDA kernel
  - Function executed on the GPU
  - Use N threads
    - hello_kernel <<< 1, N >>>();
- printf() is buffered on the GPU memory
- A kernel gives back the control to the CPU immediatly after launch
- CudaDeviceSynchronize() waits until everything on the GPU is completed

GPUs @ UBELIX

Example – Hello, World! (2/2)

- CUDA C
  - Use standard C syntax
- Name file *.cu
  - E.g: hello_world.cu
- Compile with nvcc.
  - Must load CUDA first!
  - Select compute capability
  - nvcc -arch sm_35 -o hello hello_world.cu
- Submit job using sbatch

GPUs @ UBELIX

Get help

- GPU expert @ UniBE
  - simon.grimm@csh.unibe.ch

Training Courses

UNIVERSITÄT BERN

Courses

- ScITS hosts courses 3 time a year:
  http://www.scits.unibe.ch/training/training_and_workshops/

- September 2019:
  - Intoduction to Linux for users/owners
  - HPC & UBELIX
  - Working with Containers
  - Deep Learning with MATLAB
  - Python for Programmers

Self-Education

- Lots of resources like tutorials/video courses on the internet

- UBELIX documentation:
- https://docs.id.unibe.ch/ubelix

- UBELIX job monitoring:
- https://ubelix.unibe.ch

Thank You!

For your attention

Michael Rolli, Nico Färber, IT-Services Office

Contact: grid-support@id.unibe.ch