

OOKAMI

Getting Started Guide



3/2023

<https://www.stonybrook.edu/ookami/>



Stony Brook
University



iACS INSTITUTE FOR ADVANCED
COMPUTATIONAL SCIENCE

1. [What is Ookami?](#)
2. [Logging in](#)
3. [Getting A64FX nodes](#)
4. [File System](#)
5. [Modules](#)
6. [Job scheduling](#)
7. [Compilers](#)
 - a. [Recommendations for A64FX](#)
 - b. [Arm](#)
 - c. [Cray](#)
 - d. [Fujitsu](#)
 - e. [GNU](#)
8. [MPI](#)
 - a. [Modules](#)
 - b. [Compilers](#)
 - c. [Job Submission](#)
9. [Vectorization](#)
 - a. [Flags](#)
10. [Profilers](#)
11. [What else?](#)
12. [Key Takeaways](#)

What is Ookami



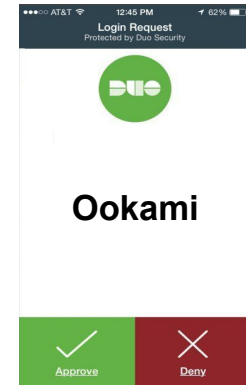
- ❑ **Testbed** providing researcher access to 176 **A64FX** nodes (48 cores each)
 - ❑ 32 GB high-bandwidth memory
 - ❑ 512 GB SSD
- ❑ Ookami also includes:
 - ❑ 1 node with dual socket **AMD Milan** (64 cores) with 512 GB memory
 - ❑ 2 nodes with dual socket **Thunder X2** (64 cores) each with 256 GB memory
 - ❑ 1 node with dual socket **Intel Skylake** Processors (36 cores) with 192 GB memory

Accessing the System



```
ssh -X NetID@login.ookami.stonybrook.edu
```

- ❑ Approve DUO prompt
- ❑ This will bring you to `login1` or `login2`
- ❑ Both are ThunderX2 - aarch64



[See FAQ entry](#)

Getting an A64FX node



- ❑ For compiling / debugging you can use the debug nodes
(those are not exclusive; multiple users can use them at the same time)
 - ❑ `ssh fj-debug1 (A64FX - aarch64)` or
 - ❑ `ssh fj-debug2 (A64FX - aarch64)`
- ❑ Or start a slurm job (see section 'Job Scheduling' slide 9)

File System



- ❑ Home directory: `/lustre/home/NetID`
- ❑ Scratch directory: `/lustre/scratch/NetID`
- ❑ Optional project directory: `/lustre/projects/group-name`

Location	Size	Backed Up?	Shareable?	Cleared?
<code>/lustre/home/<netid></code>	30GB	Yes	No	never
<code>/lustre/scratch/<netid></code>	30TB	No	No	30 days
<code>/lustre/projects/<your_group>*</code>	up to 8TB	Yes**	Yes	per request

*Project directories are granted upon request from the group's PI

**Some large project directories may not be backed up

[See FAQ entry](#)



Modules



- ❑ `module avail` lists modules on the login nodes for all architectures on Ookami.
 - ❑ `aarch64`
 - ❑ `x86_64`
 - ❑ `x86_64-GPU` (note that Ookami currently does not have GPUs)
- ❑ On all other nodes, only modules for the specific architecture of the current node are listed

Modules



- ❑ To see all modules (also for other architectures) use

```
lesiegmann@login2 scripts]$ module load all-architectures
lesiegmann@login2 scripts]$ module avail

----- /cm/local/modulefiles -----
cluster-tools/9.0  cmjob  freeipmi/1.6.4  ipmitool/1.8.18  module-git  null  openmpi/mlnx/gcc/64/4.0.3rc4  python37  slurm/slurm/19.05.7
cmd               dot    gcc/9.2.0       lua/5.3.5       module-info  openldap  python3          shared

----- /cm/shared/modulefiles -----
cm-pmix3/3.1.4  hdf5/1.10.1  hwloc/1.11.11  ucx/1.6.1

----- /lustre/shared/modulefiles/x86_64-GPU -----
cuda/toolkit/11.2          nvidia/cuda10.2/nvhpc/21.5  nvidia/cuda11.3/nvhpc-byo-compiler/21.5
gcc/11.1.0-openacc        nvidia/cuda11.0/nvhpc-byo-compiler/21.5  nvidia/cuda11.3/nvhpc-nompi/21.5
nvidia/cuda10.2/nvhpc-byo-compiler/21.5  nvidia/cuda11.0/nvhpc-nompi/21.5  nvidia/cuda11.3/nvhpc/21.5
nvidia/cuda10.2/nvhpc-nompi/21.5          nvidia/cuda11.0/nvhpc/21.5

----- /lustre/shared/modulefiles/s/x86_64 -----
all-architectures  cmake/3.21.0  git/2.29          intel/mpi/64/2020/20.0.2  pwanalyzer/0.18.2  visit/3.2.1
anaconda/3         curl/7.73.0   intel/compiler/64/2020/20.0.2  intel/tbb/64/2020/20.0.2  quantum-espresso/intel/6.8  zsh/5.8
aoacc/3.0.0        gethost/1.0   intel/mkl/64/2020/20.0.2      ncurses/6.2            template

----- /lustre/shared/modulefiles/aarch64-GPU -----
nvidia/cuda11.0/nvhpc-byo-compiler/21.5  nvidia/cuda11.0/nvhpc/21.3  nvidia/cuda11.3/nvhpc-byo-compiler/21.5  nvidia/cuda11.3/nvhpc/21.3
nvidia/cuda11.0/nvhpc-nompi/21.5          nvidia/cuda11.0/nvhpc/21.5  nvidia/cuda11.3/nvhpc-nompi/21.5          nvidia/cuda11.3/nvhpc/21.5

----- /lustre/shared/modulefiles/aarch64 -----
all-architectures  gcc/10.3.0          libfabric/1.12.1  ncurses/6.2  openmpi/gcc11/4.1.0
anaconda/3         gcc/11.1.0          libffi/3.1        ncurses/arm/gcc/6.2  openmpi/gcc11/4.1.1
archiconda/3       gethost/1.0         libffi/3.3        netcdf/4.7.4        openssl/1.1.1h
arm-modules/20     git/2.29            libgd/gcc/2.3.1  netcdf/fujitsu/4.8.0  p7zip/16.02
```

Modules



- ❑ `module load modulename` will load a module
- ❑ `module list` shows all modules you have currently loaded
- ❑ `module purge` will remove all loaded modules

[See FAQ entry](#)



Job Scheduling



- ❑ SLURM is used for job scheduling
- ❑ `man sbatch` opens the manual
- ❑ Jobs can be either
 - ❑ Interactive: You will have an interactive terminal session directly on a compute node
 - ❑ Submitted via a run script: Job will run based on the commands in the script

SLURM Partitions



Partition	Time Limit	Min Nodes	Max Nodes	CPU Architecture
short	4 hours	1	32	A64FX
medium	12 hours	8	40	A64FX
large	8 hours	24	80	A64FX
long	2 days	1	8	A64FX
extended	7 days	1	2	A64FX
milan-64core	1 day	1	1	AMD Milan
skylake-36core	1 day	1	1	Intel Skylake

[See FAQ entry](#)

Example: Interactive Job



❑ Interactive job

```
srun -N 1 -n 48 -t 00:10:00 -p short --pty bash
```

Number of nodes
Tasks per node
Time
Partition

Will get you to a compute node so you can interactively run jobs (e.g. for compiling, debugging)

[See FAQ entry](#)

Example: Job Script



```
#SBATCH --job-name=examplejob
#SBATCH --output=examplejob.log
#SBATCH --ntasks-per-node=24
#SBATCH -N 1
#SBATCH --time=00:10:00
#SBATCH -p short
module load CPE/21.03
module load cray-mvapich2_nogpu_sve/2.3.5
mpicc /lustre/projects/global/samples/HelloWorld/mpi_hello.c -o mpi_hello
srun ./mpi_hello
```

Number of nodes
Tasks per node
Time
Partition

Sbatch jobs inherit the launch environment

Execute with `sbatch file.slurm`

[See FAQ entry](#)

Useful SLURM Commands



Command	Effect
<code>man sbatch</code>	list all available options
<code>squeue</code>	lists all jobs running and waiting
<code>squeue -u <NetID></code>	lists all jobs of a user
<code>scancel <Job ID></code>	cancel a job
<code>sinfo -s</code>	list all partitions



Compilers

Available Compilers



- GNU
- Arm
- Cray
- Fujitsu
- Nvidia
- Intel (for Intel Skylake)
- AOCC (for AMD Milan)

Compiler Recommendations



- ❑ For standard C, C++, Fortran we recommend to use
 - ❑ Cray
 - ❑ Fujitsu
- ❑ For anything else
 - ❑ Arm
- ❑ Use GNU only when you have trouble porting or for comparison.
In most cases it will not give you good performance!

- ❑ Five versions available
 - ❑ 21, 21.1, 22.0, 22.0.2, 22.1
- ❑ `module load arm-modules/<version number>`

Language	Compiler Name
C	armclang
C++	armclang++
Fortran	armflang

[See FAQ entry](#)

- ❑ Three versions available

- ❑ 10.0.1, 10.0.2, 10.0.3, 15.0.1

Note that the modules are called 20.10, 21.03, 21.10, 22.03, 22.10 and 23.02 due to an inconsistency in the naming

- ❑ Separate compilers for SVE / non-SVE instructions

- ❑ CPE / CPE-nosve modules

- ❑ Loading these modules adds `/opt/cray/pe/modulefiles` to your path, which contains all the Cray-specific modules

- ❑ Cray-specific modules now show in `module avail`

[See FAQ entry](#)

- ❑ Version 10.0.1
 - ❑ module load CPE/20.10
- ❑ Version 10.0.2
 - ❑ module load CPE/21.03

Language	Compiler Name
C	cc
C++	CC
Fortran	ftn

- ❑ Version 10.0.3 (Load either)
 - ❑ module load CPE/21.10
 - ❑ module load CPE/22.03
 - ❑ module load CPE/22.10
- ❑ Version 15.0.1
 - ❑ module load CPE/23.02

[See FAQ entry](#)

- ❑ Three versions: 4.2.0, 4.5.0, 4.7.0, 4.8.0
- ❑ `module load fujitsu/compiler/<version number>`
- ❑ Comes with two “backends”: Traditional, and LLVM (C/C++ compilers only)
 - ❑ Different compiler flags come with each backend (see `man fcc_trad_mode`, `man fcc_clang_mode` for details)

Language	Compiler Name	Choosing a Backend (Default: Traditional “Mode”)
C	<code>fcc</code>	Traditional: <code>-Nnoclang</code> LLVM: <code>-Nclang</code>
C++	<code>FCC</code>	Traditional: <code>-Nnoclang</code> LLVM: <code>-Nclang</code>
Fortran	<code>frt</code>	N/A

Note: There are known issues with the Fujitsu MPI in earlier version
We recommend using version 4.8

[See FAQ entry](#)

- ❑ Several versions available
 - ❑ 7.5.0, 8.5.0, 9.4.0, 10.2.0, 10.3.0, 11.1.0, 11.2.0, 11.3.0, 12.1.0, 12.2.0
 - ❑ Note that SVE is just supported starting from version 10

- ❑ `module load gcc/<version number>`

Language	Compiler Name
C	gcc
C++	g++
Fortran	gfortran

[See FAQ entry](#)



MPI



- ❑ Two installed implementations
 - ❑ OpenMPI, MVAPICH

- ❑ Fujitsu compiler comes with its own implementation
 - ❑ Commands: `mpifcc` (C), `mpiFCC` (C++), and `mpifrt` (Fortran)

- ❑ Each compiler has its own MPI pairing -- so load the proper module!
 - ❑ i.e., use the Cray-compiled MPI with the Cray compiler
 - ❑ You can override this if you *really* know what you're doing :)

- ❑ Loading the MPI module will also load the corresponding compiler

- ❑ For Cray, load the compiler first, and then MPI (separate commands)

MPI Modules



Compiler	OpenMPI modules	MVAPICH modules
GCC	<code>openmpi/gcc<version>/<version></code>	<code>mvapich2/gcc<version>/<version></code>
ARM	<code>openmpi/arm<version>/<version></code>	<code>mvapich2/arm<version>/<version></code>
Cray	Not currently available	<code>cray-mvapich2_nogpu_sve/<version></code> (SVE) <code>cray-mvapich2_nogpu/<version></code> (non-SVE) NOTE: Cray cc uses a gcc-compiled MPI, let us know if there are any problems. Cray CC and ftn use a Cray-compiled MPI and work fine.

Fujitsu has its own MPI implementation: Comes installed with the `fujitsu/compiler` modules

MPI Compilers



Language	Compiler Name (Non-Fujitsu)	Compiler Name (Fujitsu - version 4.5.0)
C	<code>mpicc</code>	<code>mpifcc</code>
C++	<code>mpiCC/mpicxx/mpic++</code>	<code>mpiFCC</code>
Fortran	<code>mpifort (mpif77/mpif90)</code>	<code>mpifrt</code>

Job submission with MPI



- ❑ OpenMPI
 - ❑ Use `mpiexec`

- ❑ MVAPICH
 - ❑ Does not have `mpiexec/mpirun` commands, need to use `srun`
 - ❑ May have to add the `--mpi=pmi2` option

- ❑ Always check whether your job is running as expected!
 - ❑ Make sure your job is properly distributing your program across nodes, and not just running a copy of your program on each node!
 - ❑ Check this (interactively) first on a smaller test problem before submitting a large job



Vectorization

Vectorization



Vectorization is the process of converting an algorithm from operating on a single value at a time to operating on a set of values (vector) at one time.

❑ Examples for issues that could impact vectorization

❑ Loop dependencies

```
for(i=0; i<end; i++)  
    a[i] = a[i-1] + b[i-1];
```

❑ Indirect memory access (if `idx[i]` is a permutation of `i`, a pragma can be used to force the compiler to vectorize)

```
for(i=0; i<end; i++)  
    a[idx[i]] = b[i] + c[i];
```

❑ Non straight line code (if value of function not known at compile time)

```
for(i=0; i<CalcEnd(); i++)  
    if(DoJump())  
        i += CalcJump();  
    a[i] = b[i] + c[i];
```


Vectorization Flags



	Cray	Arm	GNU	Fujitsu	
Mode				Traditional	Clang (-Nclang)
Optimization	-O3	-O3 or -Ofast	-O3 or -Ofast	-Kfast	-O3
Vectorization	-h vector3	-mcpu=a64fx -armpl	-mcpu=a64fx	-KSV	-march=armv8.2-a+sve
Vectorization report	-h msgsg	-Rpass=loop-vectorize	-fopt-info-vec	-Koptmsg=2 -Nlst=t (creates a *.lst file with optimization information)	
Report on missed optimization	-h negmsgsg	-Rpass-analysis=loop-ve ctorize	-fopt-info-vec-mi ssed		
OpenMP	-h omp	-fopenmp	-fopenmp	-Kopenmp	
Debugging	-G 2	-ggdb	-ggdb	-g	
Large memory	-h pic	-mcmmodel=large	-mcmmodel=large	-mcmmodel=large	
Module	CPE/ <i>version</i>	arm-modules/ <i>version</i>	gcc/ <i>version</i>	fujitsu/compiler/ <i>version</i>	

Vectorization Performance



- ❑ Certain compiler vectorization are more optimal than others leading to performance differences.
 - ❑ Be sure to look into what can / can't be vectorized!
- ❑ Vectorization experiment shown below

	Fujitsu	Cray	Arm	GNU
Simple ($Y = 2X + 3X^2$)	✓	✓	✓	✓
Reciprocal	✓	✓	✓	✓
Square root	✓	✓	✓	✓
Exponential	✓	✓	✓	
Sin	✓	✓	✓	
Power	✓	✓	✓	

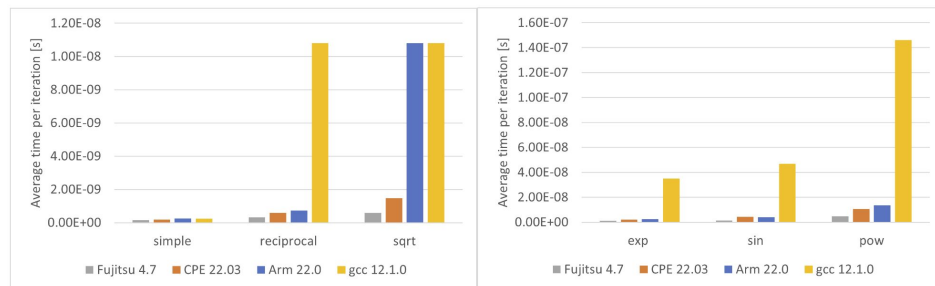


Figure 1 & 2: Runtimes of the simple math functions for different compilers.

[See FAQ entry](#)



Profilers



- ❑ TAU
 - ❑ `module load tau/2`
- ❑ CrayPAT: works only with Cray's compilers
 - ❑ Instrument a compiled binary and execute *that* to read performance metrics
 - ❑ Set up the cray programming environment, then load `perftools-base/21.12.0`
 - ❑ See `man pat_build`
- ❑ Arm FORGE suite
 - ❑ Set up ARM programming environment (e.g. `arm-modules/22.0`)
 - ❑ `module load forge/21.0.1`
- ❑ gprof (GNU profiler): does NOT work with Cray's compilers
 - ❑ Requires the “-pg” flag to be used during compilation and linking
 - ❑ 2-step process: Run the application as-is, then use gprof to collect metrics
- ❑ Fujitsu Instant/Advanced Performance Profilers (fipp/fapp): Fujitsu compilers
 - ❑ 2-step process: Use fipp/fapp to collect counters during runtime, then generate output

[See FAQ entry](#)

What else



- ❑ Get in contact!
 - ❑ Slack channel
 - ❑ **Join the Ookami office hours**
 - ❑ Tuesday, 10am - noon EDT
 - ❑ Thursday, 2pm - 4pm EDT
 - ❑ Submit a ticket <https://iacs.supportsystem.com/>
- ❑ Check the FAQ on our website <https://www.stonybrook.edu/ookami/>

Key Takeaways



- ❑ **Don't expect to get good performance immediately!**
- ❑ Test the different compilers. There can be huge performance differences.
- ❑ Don't start with the GNU compiler, just because you are used to it. It will in most cases not give the best performance!
- ❑ Check if your code is vectorized
- ❑ Choose the appropriate MPI
- ❑ Make sure you are on the right node
- ❑ Get in contact with the Ookami team. We are happy to support you!