- Remotely from the cross-compile workstation

  1. Compile the application (e.g. `my_app`) on the cross-compile workstation.

     ```
     <usr@host123> $ scpathcc -g ./my_app.c
     ```

  2. Use `salloc` to allocate and associate a node with your shell, then `ssh` to the allocated node.

     ```
     scx-m32n6:~ $ salloc -p scx-comp -w scx-m0n2
     salloc: granted job allocation 56980
     scx-m32n6:~ $ ssh scx-m0n2
     scx-m0n2:~ $
     ```

  3. Start up `gdbserver` on the allocated node:

     ```
     scx-m0n2:~ $ gdbserver :7654 ./my_app
           # specify the link over which gdb and gdbserver will
             communicate (e.g. TCP port :7654) and the program
             to debug)
     ```

  4. Start up `scgdb` on the cross-compile workstation:

     ```
     <usr@host123> $ scgdb ./my_app
     ```

  5. When `scgdb` returns the `(gdb)` prompt:

     ```
     <usr@host123> $ (gdb) set sysroot /opt/sicortex/rootfs/default
         # This is the location of the rootfs when the System ships;
           tells gdb where to find the shared libraries

     <usr@host123> $ (gdb) target remote scx-m0n2:7654
         # Establish the connection to the System
     ```

  6. Run the debugger as you would normally.

PN 2911-02 Rev. 01

SiCortex

Programming Quick Reference

For Version 4.0+ Software

For complete programming details, see the *SiCortex® System Programming Guide*.

Logging on to the Head Node

```
<usr@hostxxx>:~ $ ssh <head_node_name>
Password: <usr_password>
Last login: Tue Feb 22 09:29:30 2008 from host.xxx
scx-m32n6:~ $
```

Compiling Commands (for default 64-bit ABI mode)

- On-node

```
scx-m32-n6:~ $ pathcc -o myfile myfile.c
scx-m32-n6:~ $ pathf95 -o myfile myfile.f90
scx-m32-n6:~ $ pathCC -o myfile myfile.cpp
```

- Cross-compile workstation

```
$ scpathcc -o myfile myfile.c 1
$ scpathf95 -o myfile myfile.f90
$ scpathCC -o myfile myfile.cpp
```

- Autotools/Configure options

```
CC="scpathcc" ./configure --host=mips64el-sicortex-linux-gnu \
                          --build=x86_64-pc-linux-gnu
CC="scpathcc" make
```

- Compiling an MPI job
  - Native compile

```
$ pathcc -o my_mpiapp my_mpiapp.c -lscmpi
$ pathCC -o my_mpiapp my_mpiapp.cpp -lscmpicxx -lscmpi
$ pathf95 -o my_mpiapp my_mpiapp.f -lscmpi*

$ mpicc -o my_mpiapp  my_mpiapp.c
$ mpicxx -o my_mpiapp  my_mpiapp.cpp
$ mpif90 -o my_mpiapp  my_mpiapp.f90 -lscmpif90 -lscmpi
```

  - Cross-compile

```
$ scpathcc -o my_mpiapp my_mpiapp.c -lscmpi
$ scpathCC -o my_mpiapp my_mpiapp.cpp -lscmpicxx -lscmpi
$ scpathf95 -o my_mpiapp my_mpiapp.f -lscmpi

$ scmpicc -o my_mpiapp  my_mpiapp.c
$ scmpicxx -o my_mpiapp  my_mpiapp.cpp
$ scmpif90 -o my_mpiapp  my_mpiapp.f90 -lscmpif90 -lscmpi
```

Except for the `*mpi*` compiler scripts, to use the debug version of the MPI library, replace `-lscmpi` with `-lscmpi_debug` on the command line. For the `*mpi*` compiler scripts, use `--mpidebuglib` by itself instead.

* For Fortran programs that use the Fortran 90 MPI interface, add `-lscmpif90 -lscmpi` to the command line, or `-lscmpif90 -lscmpi_debug` to use the debug version.

Basic SLURM Commands

- Running jobs
  - Multinode jobs

```
$ srun -p <partition> -N <nodes> -n <tasks> <executable> [args]
```

  - Single-node jobs

```
$ srun -p <partition> -N 1 <executable> [args]
```

  - Single processor jobs

```
$ srun -p <partition> -N1 -n1 <executable> [args]
```

  - Batch jobs

```
$ sbatch -p <partition> <jobscript.sh>
```

- Monitoring jobs
  - Get the status of all SLURM jobs

```
$ squeue
JOBID  PARTITION  NAME    USER  ST  TIME  NODES  NODELIST
21      Test     myprog   xxx   R   0:25      1  sc1-m0n0
```

  - Get the status of individual nodes

```
$ scontrol show job 21
JobId=21 UserId=xxx (1198) GroupId=users (110) Name=myprog
Priorit=429448796 Partition=Test BatchFlag=0 AllocNode:  Sid=sc1-m0n0:8
TimeLimit=UNLIMITED ExitCode=0:0
JobState=Completed StartTime=01/05-16:05:32 EndTime=
NodeList=sc1-m0n[0,2-4] NodeListIndices= AllocCPUs=6*4 ReqProcs=24
.
.
.
```

  - Get the current status of partitions and individual nodes

```
$ sinfo
PARTITION   AVAIL  TIMELIMIT  NODES  STATE   NODELIST
test         up    infinite    1      idle    sc1-m0n0
test         up    infinite    1      alloc   sc1-m0n2
test         up    infinite    2      down*   sc1-m0n[1,3]
air_clients? up    infinite    6     alloc   sc1-m2n[0,2-5,7-26]
```

- Canceling jobs

```
$ srun -p sc1-comp1 -b /home/dev/work/myscript.sh
srun: jobid 21 submitted

$ scancel 21
$ squeue
JOBID  PARTITION  NAME    USER  ST  TIME  NODES  NODELIST
$
```

FabriCache commands

1. Copy (`scp` or `rsync`) your data and executable to one of the nodes in the `<partition>_clients` partition.
2. On the head node, make the `/tmp/fabclient/<partition>` your working directory:

```
scx-m32n6:~ $ export SLURM_WORKING_DIR=/tmp/fabclient/<partition>
```

3. Submit your job to SLURM, using the FabriCache partition:

```
scx-m32n6:~ $ srun -p <partition>_clients -N 20 ./simulation/ my_sim
```

Using the Performance Tools

- Invoking the tools
  - Multiple node runs

```
scx-m32n6:~ $ srun -p <partition> [args] toolname [args] <myapp> [args]
```

  - Single node runs

```
scx-m32n6:~ $ srun -p <partition> -N 1 toolname [args] <myapp> [args]
```

  - Single processor runs

```
scx-m32n6:~ $ srun -p <partition> -N 1 -n 1 toolname [args]
                                                    <myapp> [args]
```

- Listing available Processor Counter events

```
scx-m32n6:~ $ toolname -l   # list events the tool can count

scx-m32n6:~ $ toolname -L <event_name>
                            # Description of the specified event
```

Debugging Applications using gdb

- Locally on-node
  1. Compile the application (e.g. `my_app`) natively on the System using the `-g` option

```
scx-m32n6:~ $ pathcc -g ./my_app
```

  2. Start up `gdb` on the head node

```
$ gdb my_app  # specify the program to debug
$ gdb my_app my_app_Core  # specify the program and core file to debug
```

  3. With `gdb` running the program

```
(gdb) gcore my_app         # Create a core file for my_app
(gdb) frame [stackframe]   # Move from one stack frame to
                             another and print the current
                             or specified stack frame
(gdb) select-frame  # Move from one stack frame to another
                      without printing the frame info
(gdb) bt [args]     # Print summary of existing stack frames to
                      trace the antecedents of the current stack
                      frame

 n             # Print only the innermost n frames
 -n            # Print only the outermost n frames
 full[n, -n]   # Print the local variables too
```