How to build Detector MPI

DetectorMPI has been developed for RedHat Linux 64 bit. It has not been tested on Windows.

The steps for building and running DetectorMPI are as follows:

1. Download and Build QT. Tested w/ QT4.8.
2. Download and install QTCreator
3. Download and build MPICH version of MPI.
4. Download MPI Detector source code and QT project files.
5. Configure build environment for MPIDetector in QTCreator.
6. Build and Run MPI Detector, for single system.
7. For running on multiple computers on a network, configure computers.

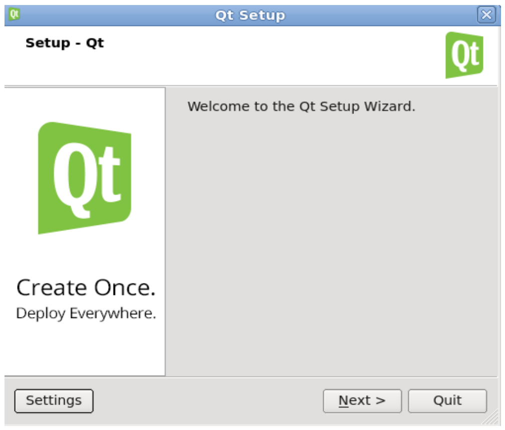
Installing QT

Download the free community version of QT from
http://www.qt.io/download-open-source/

The online downloader will download by default to your Downloads folder.

```
cd  Downloads
chmod +x qt-opensource-linux-x64-1.6.0-8-online.run
./qt-opensource-linux-x64-1.6.0-8-online.run
```

You should get the installation wizard to open:



Welcome to the Qt Setup Wizard.

Go through the wizard to install. If you do not have root, it should do a local install.

If there are errors, download source from

http://www.qt.io/download-open-source/#section-2

Installing MPI

To build you must first install the *mpich* version of MPI. This can be downloaded from

http://www.mpich.org/downloads/

Get a stable release for platform called *mpich*. DetectorMPI was first tested with mpich-3.0.4, but should work with later versions. You will be getting a tar.gz file. Expand and untar the file:

gunzip mpichxxxx.tar.gz
tar -xvf mpichxxx.tar

To make the MPI program able to run on multiple systems the MPI library and executable MPI program must be installed on an NFS shared disk. Also, the absolute path to the MPI lib and software should be the same for all systems. That is, if one system sees the software in */mnt/mpidisk/mpisoftware* then all the other systems should see the software in that path as well.

```
master $ tar xzf mpich.tar.gz
```

Create directory in mounted directory called: *mpich-install*. For example: `/mnt/nfs/mpich-install` . Then:

```
master $ cd /your/unzipped/mpi_libraries

master $ ./configure –prefix=/mnt/nfs/mpich-install 2>&1 | tee c.txt

master $ make 2>&1 | tee m.txt –j8
```

where `-jn` means make in n parallel threads. Then install it:

```
master $ make install 2>&1 | tee mi.txt
```

After installation, you should able to find mpi library under `/mnt/nfs/mpich-install`
Then, export the MPI bin path to environment variables, using editor to edit file .bash_profile:

```
master $ cd ~

master $ gedit .bash_profile
```

add the following line:

```
PATH=/mnt/nfs/mpich-install/bin:$PATH

export PATH
```

to check the environment variable. Restart the terminal, then type:

```
master $ which mpicc

master $ which mpiexec
```

The terminal should display mpich-install path.

Setting up MPIDetector Build

There are two important project files

The detectorMPI.pro file is the project setup file that lists all files part of the project and defines which compiler to use. The MPI compiler must be defined in this file as shown below.

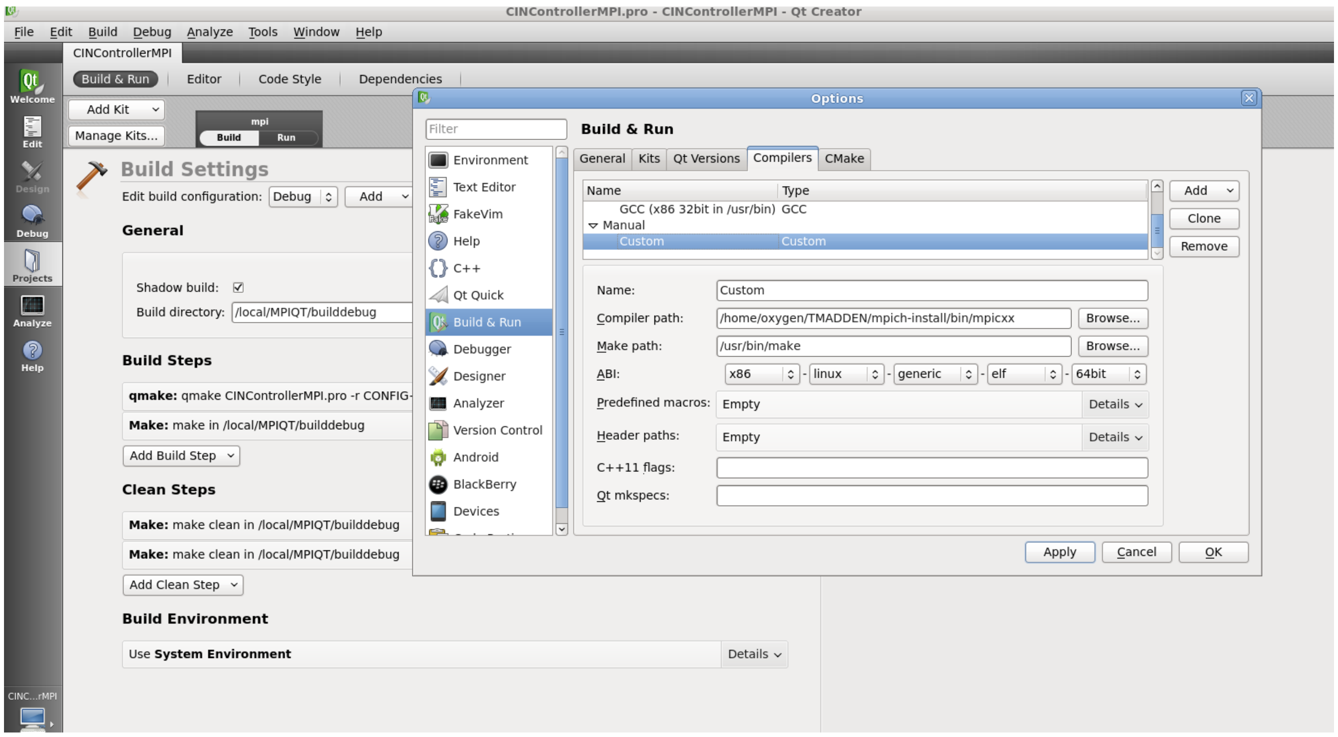
```
#-------------------------------------------------
#
# Project created by QtCreator 2011-12-08T16:30:52
#
#-------------------------------------------------
QMAKE_CXX = mpic++
QMAKE_CXX_RELEASE = $$QMAKE_CXX
QMAKE_CXX_DEBUG = $$QMAKE_CXX
QMAKE_LINK = $$QMAKE_CXX
QMAKE_CC = mpicc

QMAKE_CFLAGS += $$system(mpicc --showme:compile)
QMAKE_LFLAGS += $$system(mpicxx --showme:link)
QMAKE_CXXFLAGS += $$system(mpicxx --showme:compile) -DMPICH_IGNORE_CXX_SEEK
-DUSE_MPI
QMAKE_CXXFLAGS_RELEASE += $$system(mpicxx --showme:compile)
-DMPICH_IGNORE_CXX_SEEK
```

The other file is the detectorMPI.pro.user file that is the build configuration. This must be set up as well.

Then in QTCreator configure the build configurations. Once the build configuration is set up, you should be able to *Build All* from QTCreator. If *Build All* works, then save a copy of your .user file with *cp detectorMPI.pro.user   detectorMPI.pro.userSAVE*

You may have to add a compiler that QTCreator knows about. Go to the PROJECTS area in QTCreator, and press *Manage Kits*. There is an area to add compiler.



Edit build configuration: Debug

Shadow build: ☑

Build directory: /local/MPIQT/builddebug

qmake: qmake CINControllerMPI.pro -r CONFIG-

Make: make in /local/MPIQT/builddebug

Make: make clean in /local/MPIQT/builddebug

Make: make clean in /local/MPIQT/builddebug

Use **System Environment**

Name	Type
GCC (x86 32bit in /usr/bin)	GCC
Manual	
Custom	Custom

Name: Custom

Compiler path: /home/oxygen/TMADDEN/mpich-install/bin/mpicxx

Make path: /usr/bin/make

ABI: x86 - linux - generic - elf - 64bit

Predefined macros: Empty

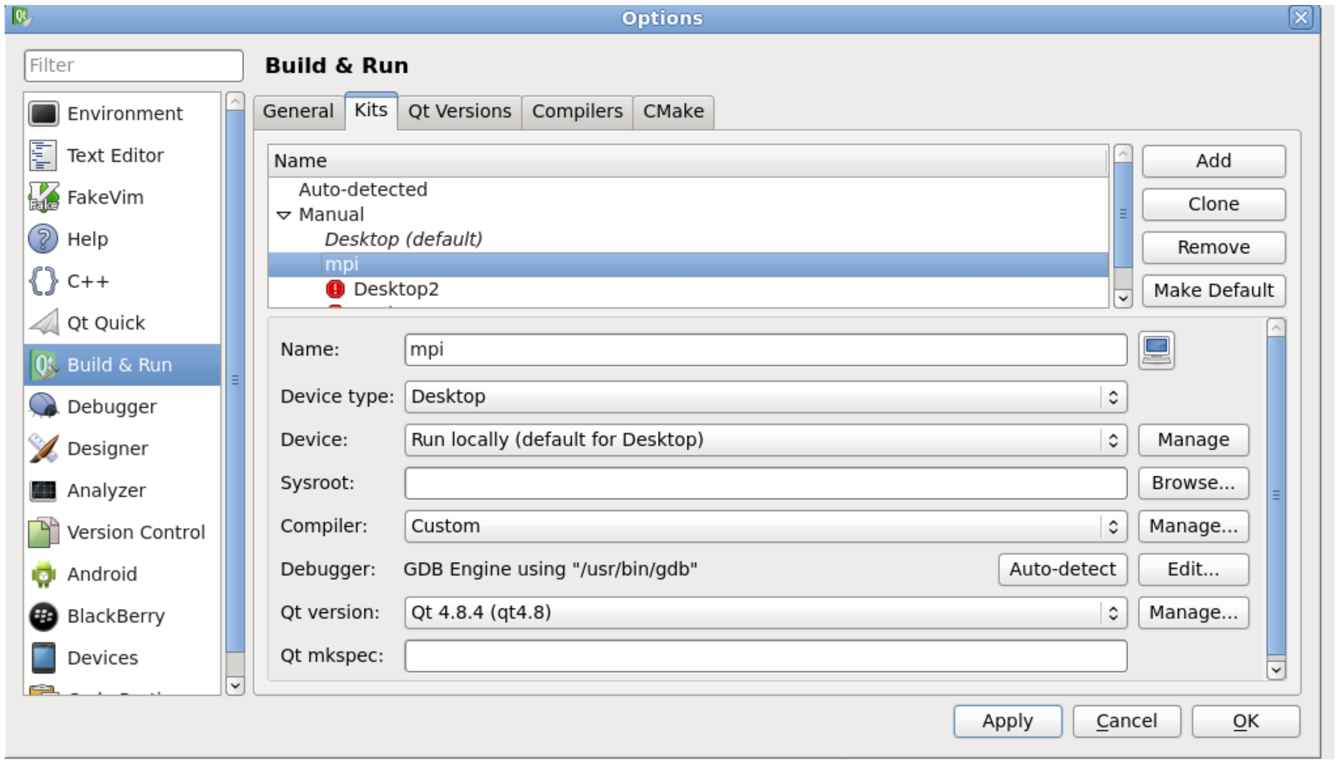
Header paths: Empty

C++11 flags:

Qt mkspecs:

Add the compiler similar to above.

You can add an mpi kit, under add kits, like below:



Build & Run

Name
Auto-detected
Manual
Desktop (default)
mpi
Desktop2

Name:	mpi
Device type:	Desktop
Device:	Run locally (default for Desktop)
Sysroot:	
Compiler:	Custom
Debugger:	GDB Engine using "/usr/bin/gdb"
Qt version:	Qt 4.8.4 (qt4.8)
Qt mkspec:	

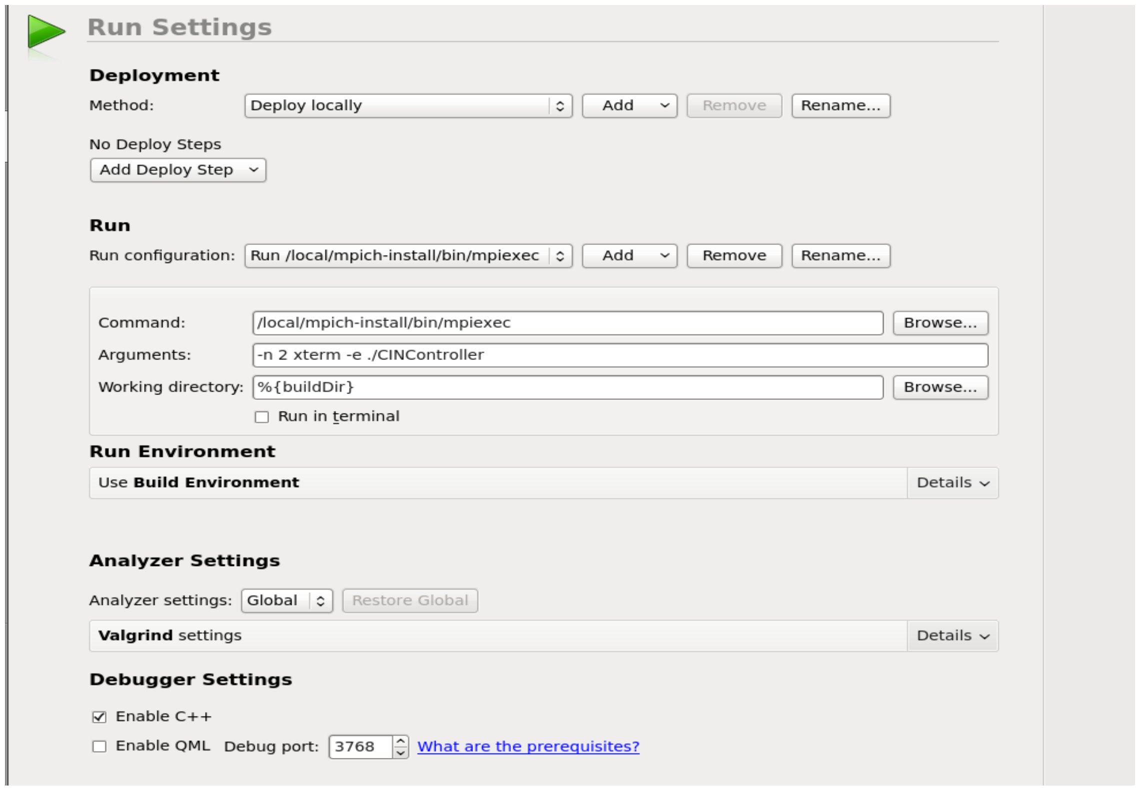
Setting up running the MPI program from QTCreator.

Running the MPI program from command is done this way:

cd build_directory

/local/mpich-install/bin/mpiexec -n 2 xterm -e ./detectorMPI

The above commands are setting pwd to the directory containing the detectorMPI binary. We then use *mpiexec* to launch the MPI program. *-n 2* means we run two copies of the process. If we set *-n 4* we would get 4 copies of the process. The *xterm -e* command forces the detectorMPI processes to run in an xterm, with each process getting an xterm. This allows each process to have its own stdin and stdout. To set this up in QTCreator, set up the RUN configuration as below.


Run Settings

Deployment

Method: Deploy locally

No Deploy Steps

Run

Run configuration: Run /local/mpich-install/bin/mpiexec

Command: /local/mpich-install/bin/mpiexec

Arguments: -n 2 xterm -e ./CINController

Working directory: %{buildDir}

☐ Run in terminal

Run Environment

Use **Build Environment**

Analyzer Settings

Analyzer settings: Global

Valgrind settings

Debugger Settings

☑ Enable C++

☐ Enable QML Debug port: 3768

Debugging detectorMPI with QTCreator

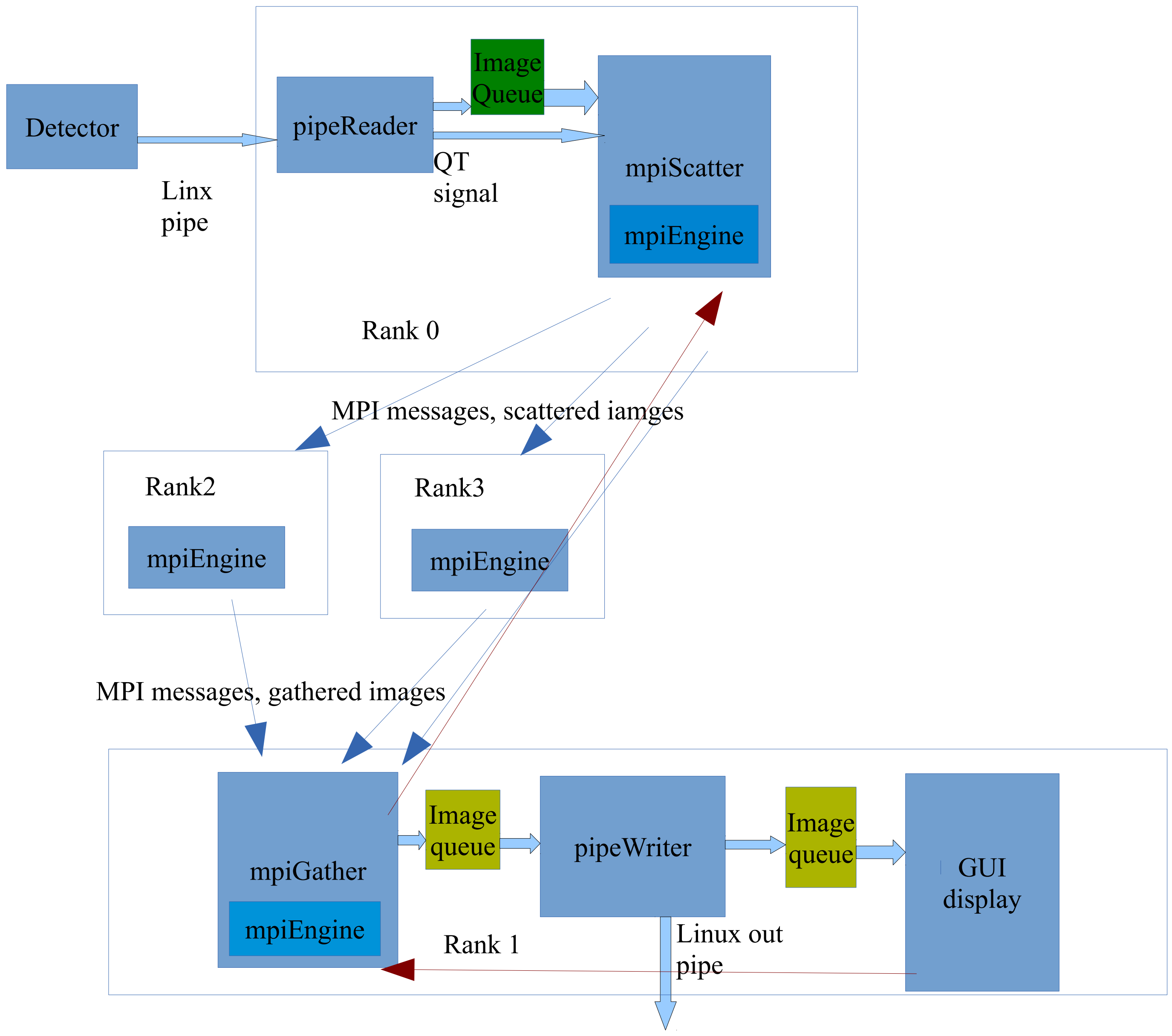
The easiest way to debug detectorMPI is to start the program running from the command line or QT Creator. This is not debugging, but just running. Then connect to the running process you wish to debug with QTCreator. To debug several processes, you can start several instances of QTCreator.

The menu on QTCreator is *Debug->StartDebugging->Connect to Running Application.*

Theory of Operation

detectorMPI is designed in a way so a programmer need not know much about MPI to do parallel processing. It is necessary to understand the basics of C++ and QT.

The model of detector MPI is shown below. We have processes called "ranks" numbered 0 to N-1. Rank 0 is where fresh detector data arrives, and is scattered to all ranks. Rank 1 is the GUI and end of the processing line, where data is stored to disk. Other ranks just process data.



Rank 0 has a *pipeReader* that can either generate test images, or read images from a detector via a linux pipe. Image data is in a binary format defined in *pipebinaryformat.cpp*. Change this file to change the format. Images from the pipe are queued on a queue, and stored temporarily. *mpiScatter* gets the images off the queue, and scatters one image per rank. So if we have 4 ranks, we scatter 4 images. If we have only one image from detector, then only rank 0 gets data to process. *mpiScatter* then broadcasts a signal to tell all ranks to process their images. When processing is done, a 2nd broadcast is sent out and rank 1 will gather all processed images from all ranks with *mpiGather*. These images are queued and sent to the *pipeWriter*, where they can be stored to the linux filesystem. Finally, the images are sent to the GUI for display. For setup of the calculations, the GUI can send messages to all MPI ranks by first sending a message to rank0. Then rank0 broadcasts the settings to all the ranks, assuring all ranks have the same GUI settings. These are the red arrows: GUI sends QT signal to *mpiEngine* inside *mpiGather* object. Then *mpiEngine* translates message to MPI and sends to Rank0. Rank 0 in turn broadcasts the message to all ranks. In this way the GUI sets up all the ranks the same way.

The Public Image

Each instance of *mpiEngine* in the group of processes has “public images.” Public images are memory spaces in each rank that can be transferred between ranks. When setting up the system, the programmer declares how large the public images are, and how many. Available are *ushort* images and *double* images. These are used for all calculations in images that must be shared between processes. If any image must be scattered or gathered, then it should be in public image memory. See code for more information.

How Dark Subtraction is done with Multiple ranks

When doing dark subtraction, each rank has a copy of a back ground image. Data images are scattered to all ranks, and rank subtracts the back ground from its image. *mpiGather* than gets all computed images and sends to the output *pipeWriter*.

It is more complicated if we must average many dark images to compute the back ground image. The code comments document this, but the basic steps are:

1. Detector spews out a total of N dark images, or more than N.
2. *mpiScatter* scatters images to all ranks, with each rank getting possibly different numbers of images, depending on how fast the detector is. For example, if we wish to average 1000 dark images, rank 0 may get 631 images and rank 1 may get 359 images. The detector may send out more than 1000 dark images. Therefore the system must keep track of this.
3. Each rank computes a sum of its collection of images. We call this a *partial sum*, because each

sum is computed on a subset of the dark images.

4. When N or more images have been scattered, all ranks send their partial sum of images to rank 1.
5. Rank 1 adds up all the partial summed images, and computes average image.
6. All ranks get a copy of the final average image, and store in own memory.

An example of this is in the code.

How to do your own MPI calculations

There are two projects released:
detectorMPI.pro, which does only dark subtraction.
XpcsMPI.pro, which does dark subtraction, thresh holding on noise, and simple image compression.
XpcsMPI.pro was created as shown below:

Make a new project based on detectorMPI

cp detectorMPI.pro xpcsMPI.pro
cp detectorMPI.pro.user xpcsMPI.pro.user

nedit xpcsMPI.pro
Find:detectorMPI   Replace: xpcsMPI

nedit xpcsMPI.pro.user
Find:detectorMPI   Replace: xpcsMPI

Now make your classes based on dark subtraction:
cp mpidarksubtract.cpp   mpixpcs.cpp
cp mpigatherdark.cpp xpcsgather.cpp
cp mpiscatterdark.cpp xpcsscatter.cpp
cp mpidarksubtract.h mpixpcs.h
cp mpigatherdark.h xpcsgather.h
cp mpiscatterdark.h xpcsscatter.h
cp mpicontrolgui.ui xpcsgui.ui
cp mpicontrolgui.h xpcsgui.h
cp mpicontrolgui.cpp xpcsgui.cpp

For the *xpcsgui.ui* file, *nedit* it, and Find/Replace to change class name to *xpcsGui*. Do the same for the *xpcsgui.h*, and *cpp* files as well. This makes a copy of the gui you can work on.

Open QTCreator

Add Existing Items to Headers and Source files to add in your new files.

Remove the older version of the files from the projects, as listed.

mpidarksubtract.cpp mpigatherdark.cpp mpiscatterdark.cpp

mpidarksubtract.h   mpigatherdark.h   mpiscatterdark.h

Make a new main() file

cp main_mpi.cpp main_xpcs.cpp

Add to the QT project. Edit it so it will make objects for the new project, and not the original dark subtract project.

Edit each of your new files and change the class names with Find/Replace to

something like *mpiXPCS*. Remember to change the #ifndef XXXX at top of the h files.

Add your new files to any repository to make sure they are saved.

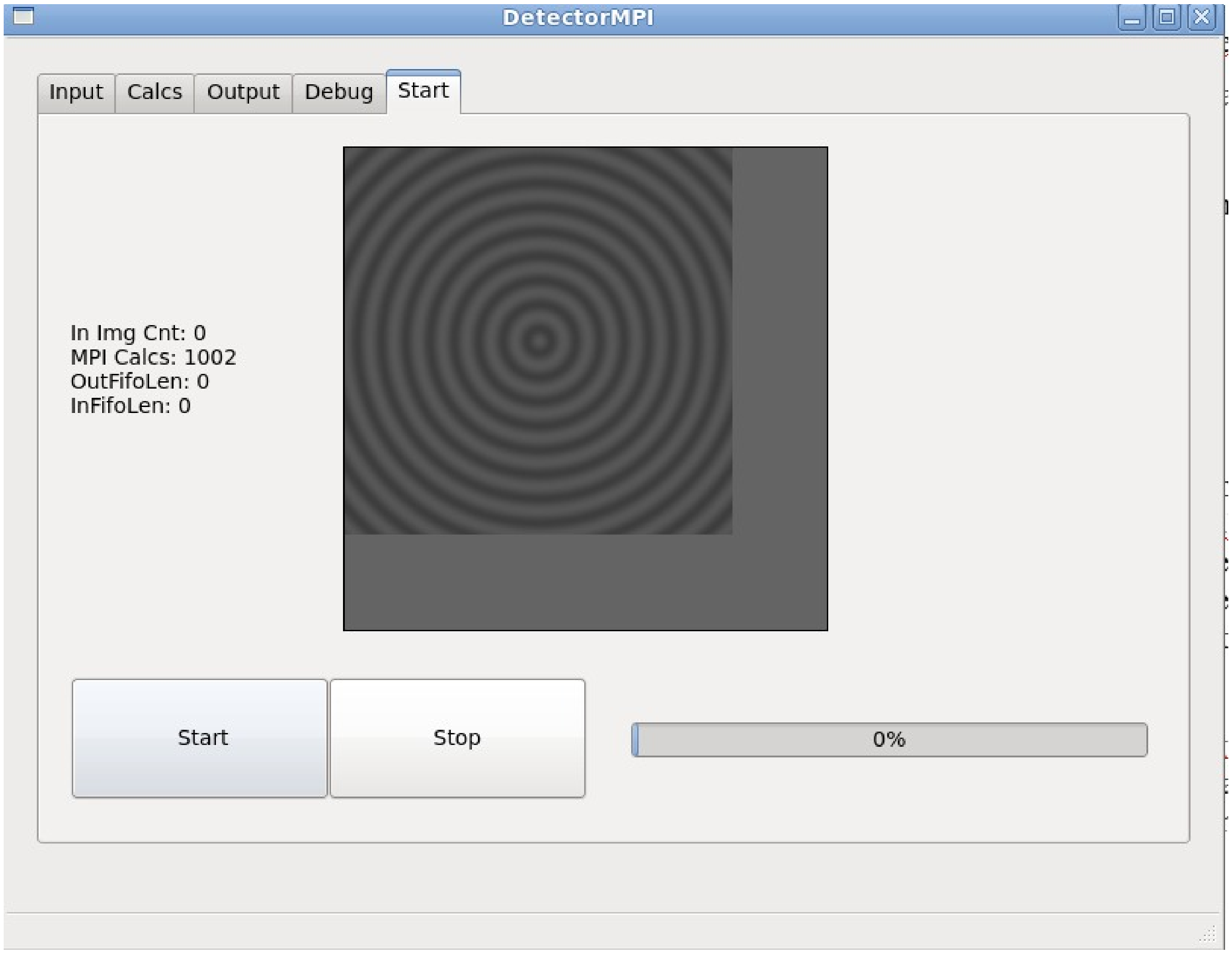
Make a GUI

The first thing to do is to design your user interface. This defines what the user inputs to the program, and what is controlled in the MPI calculations. It will help to make a list of all the things detector MPI must do.

Use QTCreator to drop buttons etc. Onto your GUI. The GUI included is a good start. It has a the following tabs:

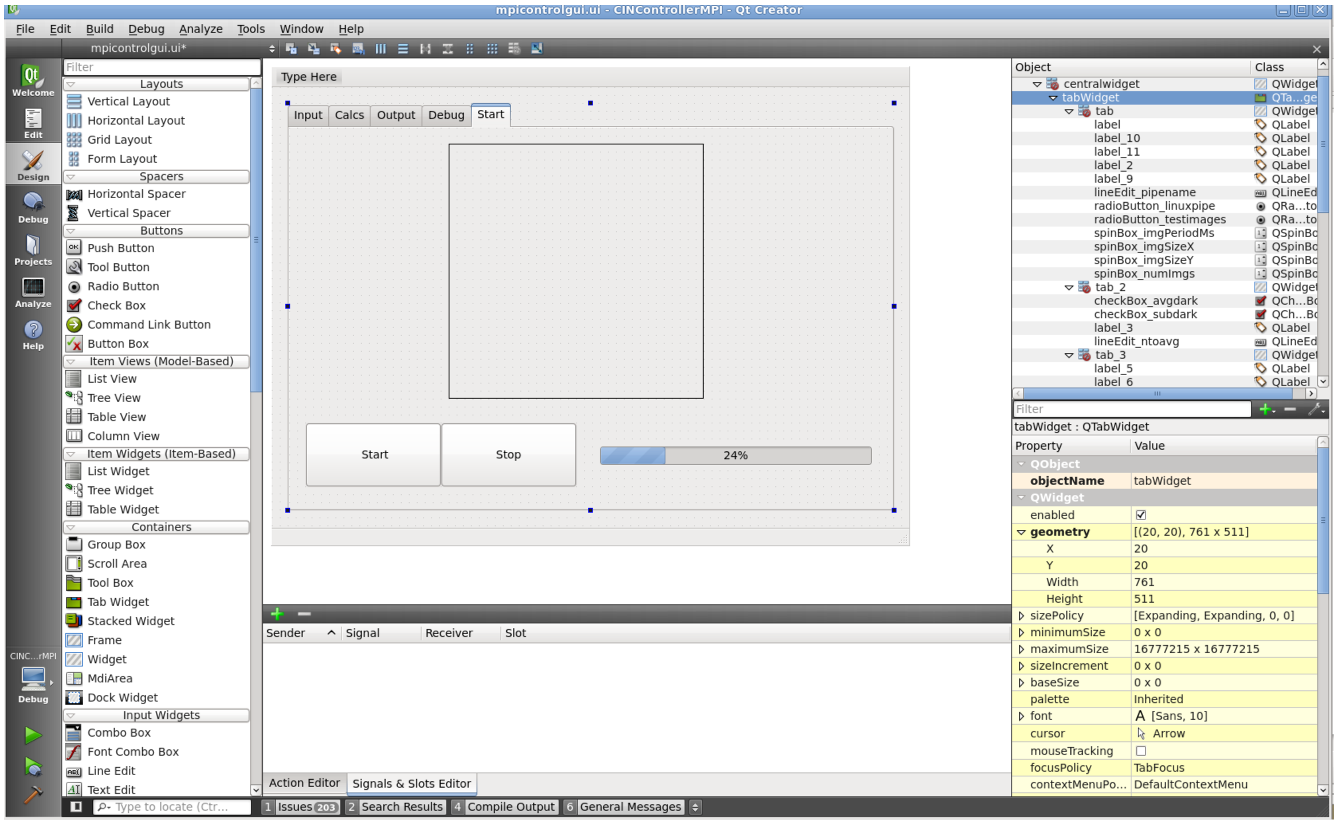
1. Input- where data originates.
2. Calcs- what calculations are done on the data in MPI
3. Output- where output data is sent after MPI calcs.
4. Debug- controls to help debug the program.
5. Start- start and stop processing and display of images.

See the QT website for info on how do develop a GUI in QTCreator.



In Img Cnt: 0
MPI Calcs: 1002
OutFifoLen: 0
InFifoLen: 0

DetectorMPI Gui



Object	Class
centralwidget	QWidget
tabWidget	QTa...ge
tab	QWidget
label	QLabel
label_10	QLabel
label_11	QLabel
label_2	QLabel
label_9	QLabel
lineEdit_pipename	QLineEd
radioButton_linuxpipe	QRa...to
radioButton_testimages	QRa...to
spinBox_imgPeriodMs	QSpinBo
spinBox_imgSizeX	QSpinBo
spinBox_imgSizeY	QSpinBo
spinBox_numImgs	QSpinBo
tab_2	QWidget
checkBox_avgdark	QCh...Bo
checkBox_subdark	QCh...Bo
label_3	QLabel
lineEdit_ntoavg	QLineEd
tab_3	QWidget
label_5	QLabel
label_6	QLabel

tabWidget : QTabWidget

Property	Value
QObject	
objectName	tabWidget
QWidget	
enabled	☑
geometry	[(20, 20), 761 x 511]
X	20
Y	20
Width	761
Height	511
sizePolicy	[Expanding, Expanding, 0, 0]
minimumSize	0 x 0
maximumSize	16777215 x 16777215
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [Sans, 10]
cursor	Arrow
mouseTracking	☐
focusPolicy	TabFocus
contextMenuPo...	DefaultContextMenu

QTCreator gui design.

The C code that correspond to the GUI is in mpiControlGui.cpp. This file has a function that executes for each button that is pressed. These functions are generated by QTCreator for you. See QT website.

Put GUI Settings into a class

Once you figure out the controls on the GUI, edit the file *signalmessage.h*. Edit the class *guiMessageFields* to reflect all values that are set up in the GUI. When you hit a button on the GUI, a value in a *guiMessageFields* object should be set. This is done by having a QT slot for each control on the GUI screen. On the GUI screen, right click on the QTCreator design screen, and *goto Slot*. In the code, set the GUI control value into the *guiMessageFields* object.

When you update the GUI, this *guiMessageFields* object will be sent as a QT signal to the MPI code, and then sent as MPI message to all MPI processes in the *detectorMPI* program. In this way, your GUI settings are sent to all processes in the MPI program.

Edit imagequeitem.h

The class *imageQueueItem* defines the image structure. It has fields like *size_x, size_y*, and a pointer to its data called imgdata. You may wish to change its data type, or add more fields and data to this class. You must also edit *imageSpecs* class.

The image queues allow the detector and MPI to run at different rates. There are two queues before the MPI calculations: a *free queue*, with empty images, and a *data queue*, with fresh images from the detector. Code contained in *pipeReader* will take items from free queue, fill it in with image data, and place on the data queue. Then *pipeReader* sends out a signal to *mpiScatter*, telling it that new images are available for the MPI processing.

Subclass of mpiScatter

The next step is to either edit *mpiscatterdark.h, cpp*, or make a subclass of *mpiScatter. mpiScatter* is a class that gets your GUI settings, then tells all MPI processes what your GUI settings are. Also, fresh images from your detector come into *mpiScatter*, and *mpiScatter* sends them to all the MPI processes, then kicks off the MPI concurrent calculations for each set of images.

In this class you will need to make code that is specific to your calculations. You will edit all the functions in *mpiscatterdarh.cpp*.

Functions you need to consider are:

gotMPIGuiSettings- which is called when gui is updated.
onDeFifo, when a new image from detector is taken from the queue an needs to be processed.
beforeDefifo, = called on new image signal, but before we dequeue anything.
afterDefifo- called after we dequeue all the new images, and just before MPI calculations are launched.

Subclass of mpiEngine

To implement your calculations you can either subclass *mpiEngine*, or edit *mpidarksubtract.cpp, h.* Your calculations will go into the function *doImgCalcs()*. Also, GUI settings must be dealt with to make sure the MPI calculations are set up based on GUI settings. This is done in *beforeFirstCalc()*. *beforeCalcs()* is called by *mpiScatter*, after images have been scattered to all process, and just before MPI processing is started.

You must also edit *signalmessage.h* to add any fields you need in the following classes:

newImgMessageFieds- a message that means a new image has come in. Image specs etc.

mpiBcastMessage- should have a *guiMessageFields* in it, so MPI gets the gui settings, also values and fields to control the MPI processing. This message is broadcast to all MPI processes during calculations.

These fields should match the code in your subclass of *mpiEngine*.

Edit mpiGather

mpiGather is a class that gathers up all the images from all the processes into one place. These images are put into a queue, and a signal is sent to the output code and GUI to save the images and display them. For simple calculations, you will not need to edit *mpiGather*. For more complex computations you may need to gather several images from each rank, say a raw image, compressed image, and perhaps FFT of image. In this case you must edit or subclass *mpiGather*.

Altering Linux pipe data format

To alter the data format of images sent over Linux pipes, edit or subclass the file *pipebinaryformat.h* and *cpp*. It is basically C file I/O to read and write to pipes. Care should be taken to deal with broken pipes, when the pipe sender process goes away. The code shows an example of this.

Testing the input and output pipes

The file *imageStreamTest.cpp* is part of the QT project, but it is not compiled with it. It is meant to be compiled as a separate program with *gcc*. This program runs on the command line and simply dumps images to stdout. Compile the program as shown in the source code comments.

Run *detectorMPI*, getting input from a named pipe. The name can be anything like */local/madpipe* shown in the GUI. You must make this pipe with

mkfifo /local/madpipe

You start the MPI code GUI by hitting *start*. The code will wait until images show up on the pipe.

Now in anther terminal you run

imgtest 1000 256 200 0> /local/madpipe

imgtest will send 1000 images of size 256x256 to */local/madpipe*. The first 200 images will be dark images, and will be type 0. The image type number allows the user to generate different types of images. The MPI software will read the images from the pipe and process.

To test the output pipe you have two options:

1. IN another terminal type

    - *mkfifo /local/madoutpipe*
    - *cat </local/madoutpipe > myfile.bin*
    - The above will read a pipe called */local/madoutpipe* and store it to a file.
    - Run the *detectorMPI* software with output pointed to a Linux pipe called */local/madoutpipe*.

2. You can run a SECOND copy of *detectorMPI*, reading input from */local/madoutpipe.*
    - In this case the first *detectorMPI* write to *madoutpipe*.
    - The 2nd copy of *detectorMPI* reads from *madoutpipe*, and displays the images.

To keep a pipe open, say for sending 1 image at a time from command lines:

sleep 9999999 > /local/madpipe &

Now when *detectorMPI* reads from that pipe, it will not close the pipe if the sender goes away. So you can do this:

imgtest 1 256 > /local/madpipe

imgtest 1 256 > /local/madpipe

imgtest 1 256 > /local/madpipe

That is, we can send one image at a time from command line. *decectorMPI* will keep reading.

You will want to kill the sleep process when done debugging.