Research and Development of an Analysis Framework for Hyper Suprime-Cam

Sogo Mineo
Department of Physics, University of Tokyo

January 21, 2010

Abstract

To explore the nature of dark energy, a new generation prime-focus camera on the Subaru Telescope, Hyper Suprime-Cam, has been developed. It has a large field of view ($\sim$1.5 degrees in diameter,) and produces $\sim$1G pixels of image per exposure. The field of view is 7 times, and the data size is 10 times, larger than those of the current prime-focus camera of Suprime-Cam.

We have developed a new analysis framework for such large amounts of data in collaboration with the Belle II group at KEK. The framework handles multiple processes on multiple computers, and controls data flow among the processes. We then ported an existing analysis pipeline used for Suprime-Cam to the framework for Hyper Suprime-Cam, and tested its performance.

We report the newly developed framework, the pipeline running in parallel on multiple machines, and its performance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Dark energy is a mysterious energy that accelerates the expansion of the universe, which has become one of the most puzzling issues in the last decade. In 1998, two groups reported the acceleration of the cosmic expansion by observing Type Ia supernovae [16][14]. The precise map of the cosmic microwave background by the Wilkinson Microwave Anisotropy Probe (WMAP) brought out the composition of the universe and revealed that dark energy accounts for as much as 73% of the total cosmic energy density while dark matter for 23% and atoms for 4.6% [10].

There are many candidates for the origin of dark energy. The cosmological constant Λ is the foremost among them. It causes negative pressure $p = -\rho$ where ρ is its constant energy density, hence $w \equiv p/\rho = -1$. Λ is deduced from vacuum energy, the energy of the zero-point quantum fluctuation of the space. The value of Λ expected from the quantum field theory, however, far exceeds the measured one by 121 orders of magnitude. Other possibilities are also suggested, such as the existence of unknown scalar fields, some uncharted nongravitational interactions, deviation from general relativity, and so on. Though it is still unknown what the dark energy is, it must be a key to new physics.

Weak gravitational lensing is a useful tool to probe the nature of dark energy. Dark energy has a peculiar feature that it is repulsive in contrast with matter that is attractive with universal gravity. Its nature therefore influences the time evolution of the universe and of the distribution of matter. The matter bends light from galaxies on the way to the earth due to general relativity, producing slightly distorted images of the galaxies on the celestial sphere. A three-dimensional map of matter distribution can be reconstructed from this "weak gravitational lensing effect," and the map is used to probe the nature of dark energy including its time variation. The weak gravitational lensing method provides an unbiased way to map the matter distribution because it does not rely on luminous–dark matter bias [15].

We need images of galaxies with high precision and in large numbers to extract the weak lensing effect due to its minuteness. It affects apparent ellipticity of galaxies only by $\sim$1%. Suprime-Cam (SC), a camera mounted on the prime focus of the Subaru Telescope produces clear images with their point-spread function of less than 0.7 arcsec in FWHM, but its field of view is not large enough to obtain the many galaxy images required to constrain the nature of dark energy. Hyper Suprime-Cam (HSC) was therefore proposed as the upgrade of Suprime-Cam. Its field of view is 1.5 degrees in diameter, $\sim$7 times the size of SC. It surveys $\sim$2000 square degrees of the sky in 150 nights.

HSC produces a large amount of image data. There are 116 CCDs laid out on the focal plane, and the total number of pixels is $\sim 10^9$. The data size produced by a shot is 2GB, 10 times larger than that in SC. An automated on-line data analysis system, which did not exist for SC, is necessary for HSC to monitor so many images. Development of efficient on-line analysis pipelines is thus important in the HSC project.

We have developed a new application framework to help the implementation of analysis pipelines for HSC's fast data processing, which are used for both on-line and off-line analyses, in collaboration with the Belle II group at KEK. The framework, PBASF, manages multiple processes on multiple computers, connecting parallel analysis modules. PBASF also utilizes Python for users to develop analysis modules efficiently.

In this thesis, we report the design and implementation of PBASF. We then describe the porting of an existing analysis pipeline onto the framework. The pipeline is for a prototype of the on-line user-assistance system for observation with HSC being developed in the National Astronomical Observatory of Japan. Finally we describe the results of the performance test of the developed pipeline, and the future prospects for the HSC analysis framework.

Chapter 2

Hyper Suprime-Cam

In order to Explore the nature of dark energy, Hyper Suprime-Cam (HSC), a new prime-focus camera of the Subaru Telescope has been developed and a wide field survey is planned in the National Astronomical Observatory of Japan (NAOJ). HSC produces 10 times as large data as Suprime-Cam (SC), the current prime-focus camera.

In this chapter we explain the physics motivation of this survey, and introduce the HSC experiment and its analysis requirements.

2.1 Physics motivation

2.1.1 Dark energy

The homogeneous and isotropic universe is described by the two equations:

$$\left(\frac{\dot{a}}{a}\right)^2 = \frac{8\pi G\rho}{3} - \frac{K}{a^2} + \frac{\Lambda}{3}, \tag{2.1}$$

$$\frac{\ddot{a}}{a} = -\frac{4\pi G}{3}(\rho + 3P) + \frac{\Lambda}{3} \tag{2.2}$$

where a is the scale parameter, ρ and P are the energy density and the pressure of the universe, K is the curvature of the space, and Λ is the cosmological constant. The cosmological constant can be considered as ideal fluid with the energy density and the pressure expressed as:

$$\rho_\lambda = \frac{\Lambda}{8\pi G}, \quad P_\lambda = -\frac{\Lambda}{8\pi G}. \tag{2.3}$$

Including ρ_λ and P_λ in ρ and P respectively, (2.2) is simplified as

$$\frac{\ddot{a}}{a} = -\frac{4\pi G}{3}(\rho + 3P). \tag{2.4}$$

Then ρ and P are the sum of components each with the equation of state $P_i = w_i \rho_i$ where w_i is a coefficient intrinsic to the component. Matter has $w = 0$, radiation has $w = 1/3$, and the cosmological term $w = -1$.

In late 1990's the expansion of the universe is revealed to be accelerating by the observation of Type Ia supernovae and the fluctuation of the cosmic microwave background. We see in (2.4) that acceleration needs a component with $w < -1/3$. We call such component "dark energy." The cosmological constant is a candidate of dark energy because it has $w = -1$ due to (2.3). One model to explain the universe though the cosmological constant Λ ($w = -1$) and cold dark matter ($w = 0$) is called the Λ-CDM model. Indeed, recent observations support the Λ-CDM model [10]. Weak lensing is another way of examining dark energy, and we introduce it in the next section.

2.1.2 Weak lensing

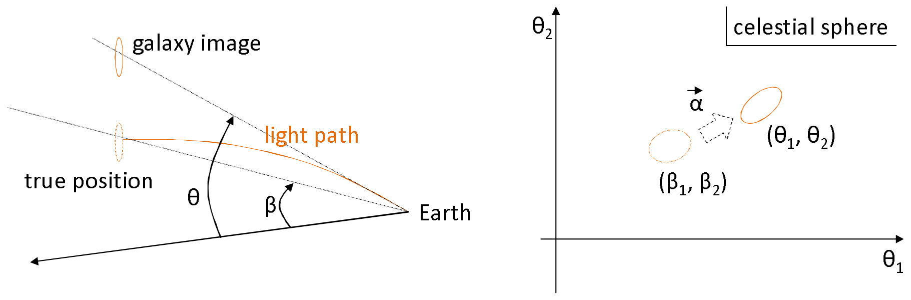


Figure 2.1: Gravitational lensing

A path of light is bent by the Newtonian potential on the way to the earth according to the general theory of relativity. Because of this gravitational lensing effect, the light source actually at angle $\boldsymbol{\beta}$ is apparently seen at angle $\boldsymbol{\theta}$ on the celestial sphere (Figure 2.1.)

$$\boldsymbol{\beta}(\boldsymbol{\theta}) = \boldsymbol{\theta} - \boldsymbol{\alpha}(\boldsymbol{\theta}). \tag{2.5}$$

When the center of a galaxy is actually at $\boldsymbol{\beta}$ and is seen at $\boldsymbol{\theta}$, its component at $\boldsymbol{\beta} + \delta\boldsymbol{\beta}$ is seen at $\boldsymbol{\theta} + \delta\boldsymbol{\theta}$ where

$$\begin{aligned} \delta\boldsymbol{\beta} &= \left(1 - \frac{\partial \boldsymbol{\alpha}}{\partial \boldsymbol{\theta}}\right) \delta\boldsymbol{\theta} \\ &\equiv A(\boldsymbol{\theta}) \delta\boldsymbol{\theta}. \end{aligned} \tag{2.6}$$

Because of the Jacobian matrix $A(\boldsymbol{\theta})$, the image of the galaxy is seen distorted around $\boldsymbol{\theta}$. We assume that the shape of the galaxy would be seen as an ellipse if no lensing effect existed. When $A(\boldsymbol{\theta})$ is represented as

$$A(\boldsymbol{\theta}) = \begin{pmatrix} 1-\kappa-\gamma_1 & -\gamma_2 \\ -\gamma_2 & 1-\kappa+\gamma_1 \end{pmatrix} = (1-\kappa) - \begin{pmatrix} \gamma_1 & \gamma_2 \\ \gamma_2 & -\gamma_1 \end{pmatrix} \tag{2.7}$$

then κ, convergence, becomes the factor that changes the size of the image, and $\gamma = \gamma_1 + i\gamma_2$, shear, the factor that changes the shape (ellipticity) of the image. We call $e = \mathrm{e}^{2i\phi}(a^2-b^2)/(a^2+b^2)$ "ellipticity," where a and b are the long and the short axis of the ellipse, and ϕ is the inclination angle of the long axis. Then the ellipticity we observe $e^{(\mathrm{obs})}$ and the ellipticity without gravitational lensing $e^{(\mathrm{s})}$ have the following relation given $\kappa \ll 1$ and $|\gamma| \ll 1$:

$$e^{(\mathrm{obs})} = e^{(\mathrm{s})} + 2\gamma. \tag{2.8}$$

$e^{(\mathrm{s})}$ is random, its average being zero, and we can reconstruct γ even if it is small, averaging $e^{(\mathrm{obs})}$ in a region:

$$\langle e^{(\mathrm{obs})}\rangle = 2\langle\gamma\rangle + O\left(\frac{\sigma_e}{\sqrt{N}}\right). \tag{2.9}$$

$\sigma_e \sim 0.2$ or 0.3 is the standard deviation of $e^{(\mathrm{s})}$. N, the number of galaxies we average, therefore has to be at least of order $O(10)$ if we want to measure $\gamma \sim 1\%$.

Now the Fourier transformed γ and κ have the same 2-point correlations and they are equivalent to the power spectrum of κ:

$$\langle\hat{\gamma}(\boldsymbol{\ell})\hat{\gamma}^*(\boldsymbol{\ell}')\rangle = \langle\hat{\kappa}(\boldsymbol{\ell})\hat{\kappa}^*(\boldsymbol{\ell}')\rangle = (2\pi)^2\delta(\boldsymbol{\ell}-\boldsymbol{\ell}')P_\kappa(\ell). \tag{2.10}$$

The power spectrum is then related to the cosmological parameters because it is written as

$$P_\kappa(\ell) = \frac{9H_0^4\Omega_m^2}{4}\int_0^\infty \mathrm{d}\lambda \frac{g^2(\lambda)}{a^2(\lambda)} P_\delta\left(\frac{\ell}{r(\lambda)}, \lambda\right), \tag{2.11}$$

$$g(\lambda) = \int_\lambda^\infty \mathrm{d}\lambda' p(\lambda') \frac{r(\lambda'-\lambda)}{r(\lambda')} \tag{2.12}$$

where λ is a comoving radial coordinate and $r(\lambda)$ is a comoving coordinate. P_δ is the power spectrum of density fluctuation $\delta(\boldsymbol{x},t) = (\rho(\boldsymbol{x},t)-\bar{\rho}(t))/\bar{\rho}(t)$. $p(\lambda)$ is the distribution of background galaxies and $g(\lambda)$ strongly depends on dark energy. This weak lensing and other methods like Type Ia supernovae technique, galaxy cluster counting, and baryon acoustic oscillations, are complementary to each others. We will obtain more precise cosmological parameters, by combining these methods [9].

2.2 Hyper Suprime-Cam

2.2.1 The Subaru Telescope and Suprime-Cam

The Subaru Telescope (Figure 2.2) of the National Astronomical Observatory of Japan (NAOJ) is located at the summit of Mauna Kea on the island of Hawaii.

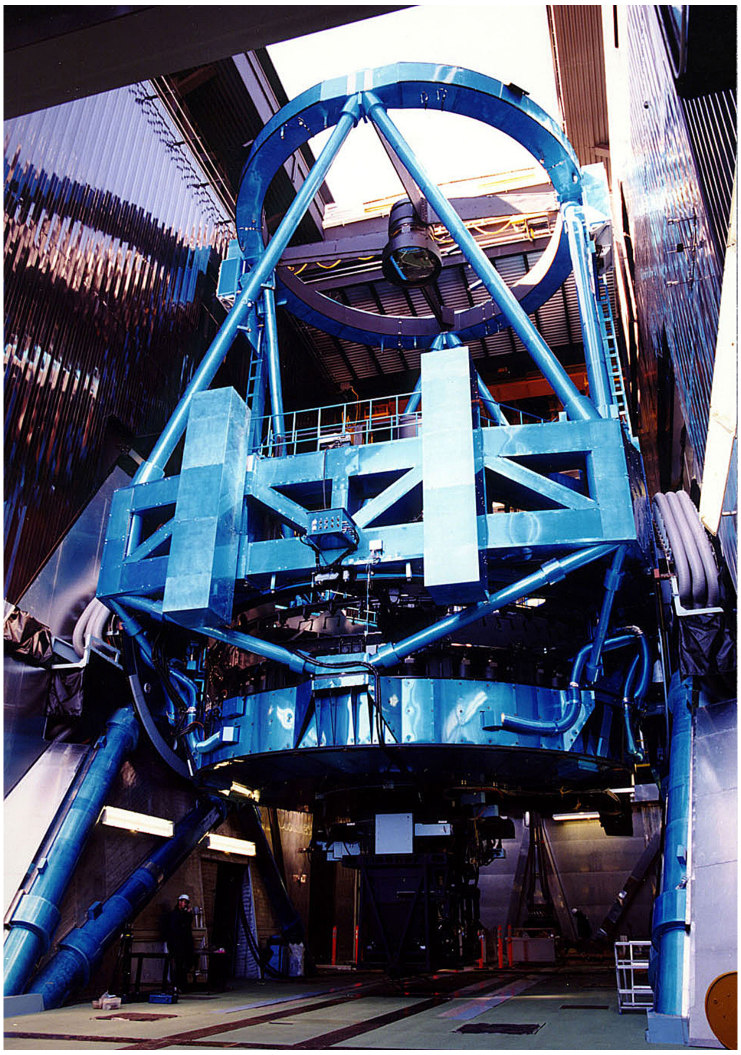


Figure 2.2: The Subaru Telescope

It is a large optical infrared telescope with a big 8.2-meter primary mirror, and with precise mechanics and optics as well.

The Subaru Telescope produces images of high resolution. For its sake, the big primary mirror is polished very smoothly and controlled by many actuators below. The shape of the dome reduces the turbulent air flow. The firm structure of the telescope driven by linear motors enables the tracking of celestial objects with high accuracy.

The telescope is unique among 8-meter types in that it has a prime focus camera, Suprime-Cam, and is capable of making observations at the prime focus with a wide field of view (FOV). There are 10 CCDs laid out on the camera, covering 34 arcmin $\times$ 27 arcmin FOV, almost the same size as the full moon. Each of the CCDs has 2k $\times$ 4k pixels.

2.2.2 Hyper Suprime-Cam

For the weak lensing survey, the FOV of Suprime-Cam is not wide enough. The surveyed area should be 2000 square degrees with four passbands (g,r,i,z.) The survey depth should be $\gtrsim$ 26th AB magnitude in g-, r-, and i-band and $\sim$ 25th AB magnitude in z-band. Then required exposure time for each pointing is about an hour in total. In order to carry out such a survey in a reasonable time, we need the survey speed to be 10 times faster. The FOV of the new camera should hence be this many times larger than that of Suprime-Cam.

Hyper Suprime-Cam (HSC) is the camera being developed to meet the above requirements (Figure 2.3.) Its FOV is 1.5 degrees in diameter and 116 CCDs are laid out in a circle (Figure 2.4.) We use CCDs of the same size as of SC, but the number of CCDs is decupled.

The CCDs for HSC is newly developed by Hamamatsu Photonics K.K. They are fully-depleted CCDs. Their thick depletion layers of 200 μm doubled the sensitivity of the CCDs at long wavelengths around 1 μm. In addition, the CCDs are four-side buttable, *i.e.* they have no cables protruding from their sides and can be placed closely at any sides of each others. The CCDs have been mounted on SC since July 2008 (Figure 2.5.)

In HSC with 116 CCDs, the total number of pixels amounts to $\sim$ 1G pixels per exposure. The tenfold increase of CCDs requires a new analysis system during observations, as described in the next section.

2.3 Analysis system

2.3.1 Current Suprime-Cam data analysis

HSC is the upgrade of SC in its field of view, and analyses in SC are also relevant to the case of HSC. There are 10 CCDs laid out on the focal plane of SC as is shown in Figure 2.6, and we introduce some terms used in describing the data analyses:

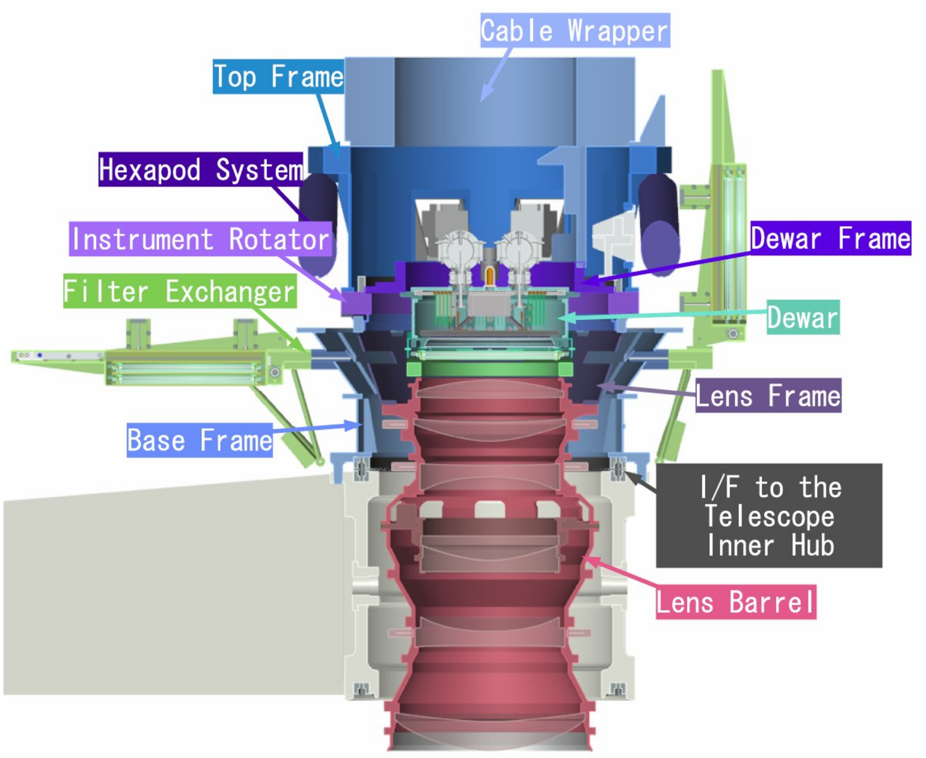


Figure 2.3: Hyper Suprime-Cam

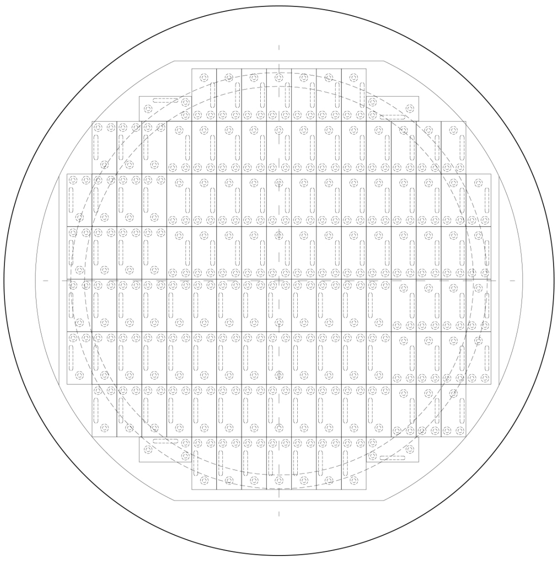


Figure 2.4: The layout of CCDs in HSC

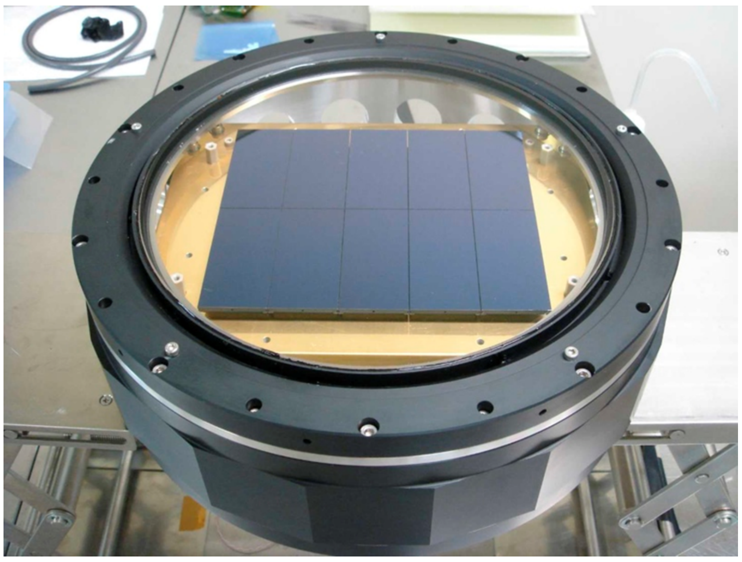


Figure 2.5: New CCDs on Suprime-Cam

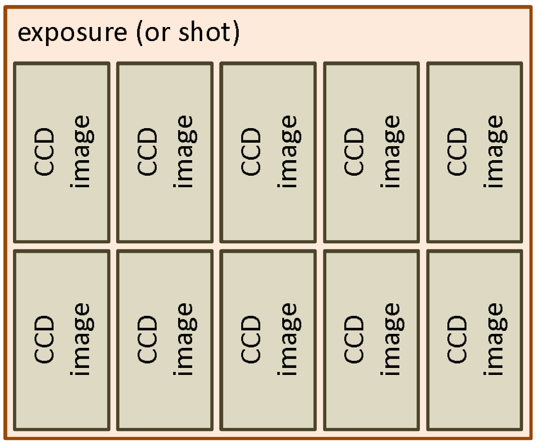


Figure 2.6: Exposures and CCD images

- A “CCD image” or a “chip” stands for an image or image file taken with one 2k $\times$ 4k CCD.
- An “exposure” or a “shot” represents a set of 10 CCD images taken simultaneously with the 10 CCDs.

Typical series of common analysis pipelines for SC are shown in Figure 2.7. These are applied to images off line, usually. The pipeline 1a is a series of primary treatments applied to each CCD images. The images came from the celestial sphere, passing through several distortive factors: the air, filters, CCD sensors, and electric read-out circuits. The pipeline 1a inversely transforms the effects of these factors.

The pipeline 1b is a mosaicking procedure. SC and HSC use many CCD sensors to cover their field of view. We have to collect many exposures of CCD images at slightly stirred positions in the celestial sphere, and to perform pattern matching of celestial objects in the images so as to superpose and co-add the images into a large mosaic. This procedure takes a long time. In HSC analyses, it will be a problem.

The pipeline 2 measures and catalogs parameters of celestial objects and perform photometric and astrometric calibrations so as for further analysis steps.

2.3.2 Data analysis for Hyper Suprime-Cam

The pipelines above have been used successfully in many off-line analyses with the Suprime-Cam data. In Hyper Suprime-Cam, however, the data rate is tenfold. We cannot compete with that amount of data using the pipelines only.

First, we have difficulty in searching for appropriate images from a huge sea for off-line analyses. We need a “tag,” or an informative header attached to

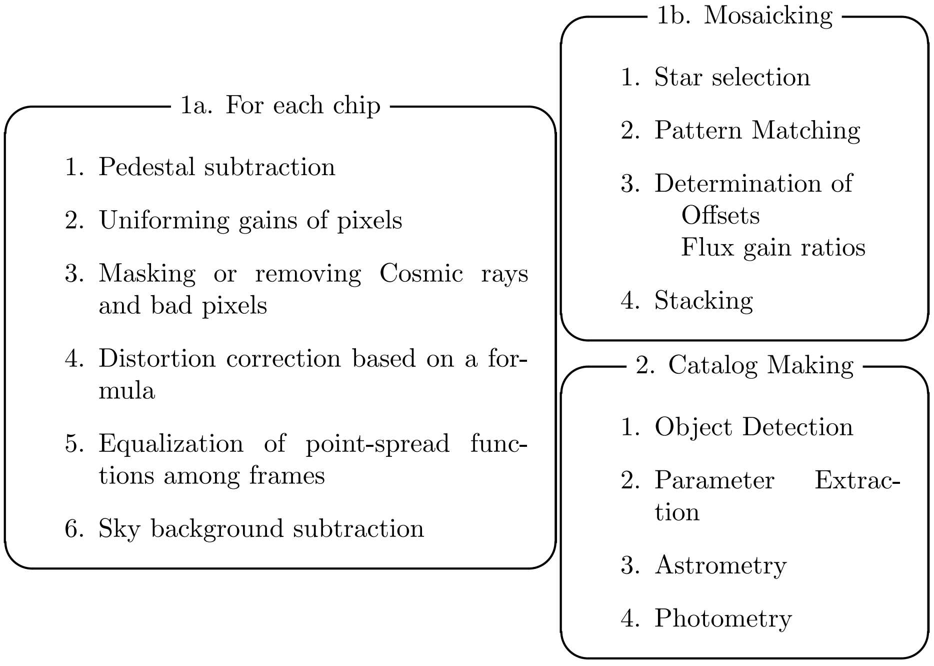

1a. For each chip
1. Pedestal subtraction
2. Uniforming gains of pixels
3. Masking or removing Cosmic rays and bad pixels
4. Distortion correction based on a formula
5. Equalization of point-spread functions among frames
6. Sky background subtraction
1b. Mosaicking
1. Star selection
2. Pattern Matching
3. Determination of Offsets Flux gain ratios
4. Stacking
2. Catalog Making
1. Object Detection
2. Parameter Extraction
3. Astrometry
4. Photometry

Figure 2.7: Typical data analysis procedures for Suprime-Cam

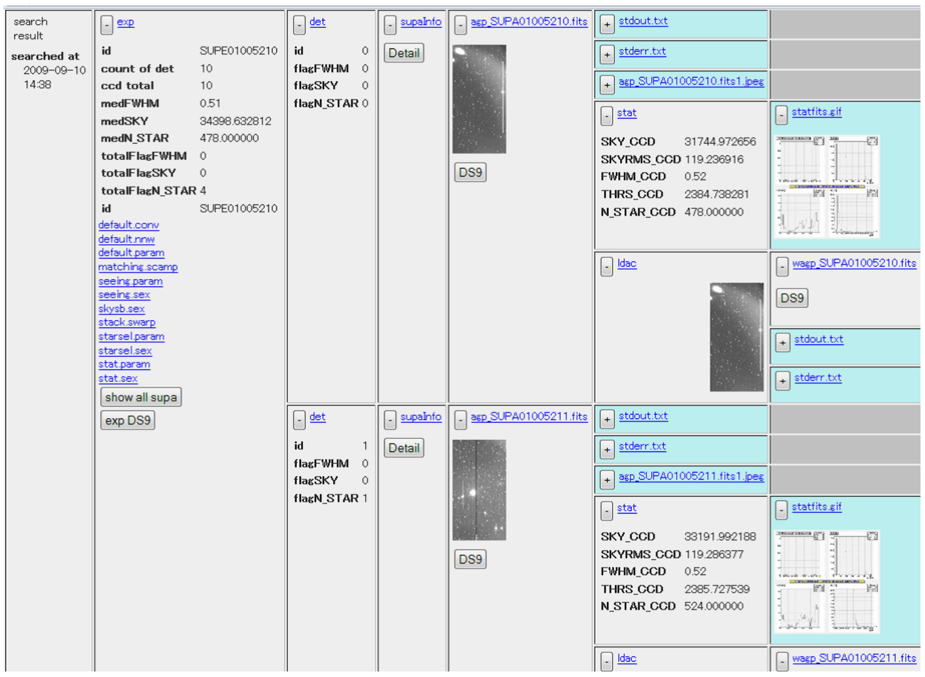
every images. We perform pre-analyses on line to write their results on the tag: blur, sky background fluctuation, the number of celestial objects in the image, etc. With the tags, we can pick out images that meet some specific criteria. We thus need a system that analyzes images coming from the camera in real-time to make such tags.

Second, the on-line analyses need a workflow controlling system. We do not have an automated workflow controller in Suprime-Cam. We instead manually checked the qualities of images coming from the camera. Suprime-Cam has only 10 CCDs, and we have only 10 images to check per exposure. HSC on the other hand produces $\sim$100 CCD images per exposure. They are more than humans can handle.

2.3.3 Prototype of on-line analysis system for HSC

NAOJ has developed a prototype of such system and tests the prototype with Suprime-Cam. The system assists users in observation and controls on-line analysis pipelines. It is controlled by the RCM (R&D Chain Management) System Software developed by QUATRE-i SCIENCE [1]. RCM consists of RCM-Web (Web user interface), RCM-Controller (Analysis controller), and RCM-DB (XML database). The prototype of the user-assistance system thereby has a web-based user interface (Figure 2.8.)

The system is installed on 8 computers. Three of them are for RCM (Web,


search result

searched at 2009-09-10 14:38

- exp

id	SUPE01005210
count of det	10
ccd total	10
medFWHM	0.51
medSKY	34398.632812
medN_STAR	478.000000
totalFlagFWHM	0
totalFlagSKY	0
totalFlagN_STAR	4
id	SUPE01005210

default.conv
default.nnw
default.param
matching.scamp
seeing.param
seeing.sex
skysb.sex
stack.swarp
starsel.param
starsel.sex
stat.param
stat.sex

show all supa
exp DS9

- det

id	0
flagFWHM	0
flagSKY	0
flagN_STAR	0

- supaInfo
Detail

- agp_SUPA01005210.fits
DS9

+ stdout.txt
+ stderr.txt
+ agp_SUPA01005210.fits1.jpeg

- stat

SKY_CCD	31744.972656
SKYRMS_CCD	119.236916
FWHM_CCD	0.52
THRS_CCD	2384.738281
N_STAR_CCD	478.000000

- statfits.gif

- ldac

- wagp_SUPA01005210.fits
DS9
+ stdout.txt
+ stderr.txt

- det

id	1
flagFWHM	0
flagSKY	0
flagN_STAR	1

- supaInfo
Detail

- agp_SUPA01005211.fits
DS9

+ stdout.txt
+ stderr.txt
+ agp_SUPA01005211.fits1.jpeg

- stat

SKY_CCD	33191.992188
SKYRMS_CCD	119.286377
FWHM_CCD	0.52
THRS_CCD	2385.727539
N_STAR_CCD	524.000000

- statfits.gif

- ldac

- wagp_SUPA01005211.fits

Figure 2.8: Web-based user interface for the user-assistance prototype

Controller, and DB) and the rest 5 are for analyses (Table 2.1.) The web server receives users' requests. The control server then controls the analysis servers and accesses the database server.

Table 2.1: Computers used for the user-assistance system prototype

Web server	Intel Xeon quad core 2.6GHz×2 / Mem 4GB
Control server	Intel Xeon quad core 2.6GHz / Mem 4GB
DB server	AMD dual core Opteron 2.8GHz ×2 / Mem 16GB
Analysis servers	Intel Xeon quad core 2.6GHz / Mem 4GB × 2
	Intel Xeon quad core 2.6GHz ×2 / Mem 4GB × 3

There are several analysis pipelines in the prototype according to Figure 2.7. We describe them below.

Real-time pipeline

When this pipeline is selected in the web screen of RCM, the RCM system launches two processes of the following RCM sub-workflow on each of the 5 analysis servers. Ten processes are invoked as a result. Suprime-Cam has 10 CCDs, and each of the 10 processes analyzes one CCD image per exposure. The launched sub-workflow is:

1. Wait for a CCD image to arrive from the camera.
2. Invoke a real-time analysis corresponding to the pipeline '1a' in Figure 2.7. The pipeline uses a framework called ROOBASF (Section 3.1). The framework enables analysis modules to transfer intermediate products *on memory* (Figure 2.9.) It omits disk I/O for intermediate products, making analyses faster. The analysis details are given in Section 4.2.
3. Go to 1.

The analysed data and tags are displayed on the screen, as shown in Figure 2.8, and stored in the database at the same time. The analysis runs in semi-real-time with exposures. Information tags produced here are useful in off-line analyses.

Each analysis for an exposure has to be done in ∼25 sec to catch up with the quickest observation speed among those supposed. To achieve this speed, we need to use many CPUs. Currently, in the prototype, we simply launch 10 independent ROOBASF processes on 5 computers to let them analyze data in parallel.

However, the ROOBASF system has difficulty in our making analysis modules run in parallel and communicate among them. We have therefore developed

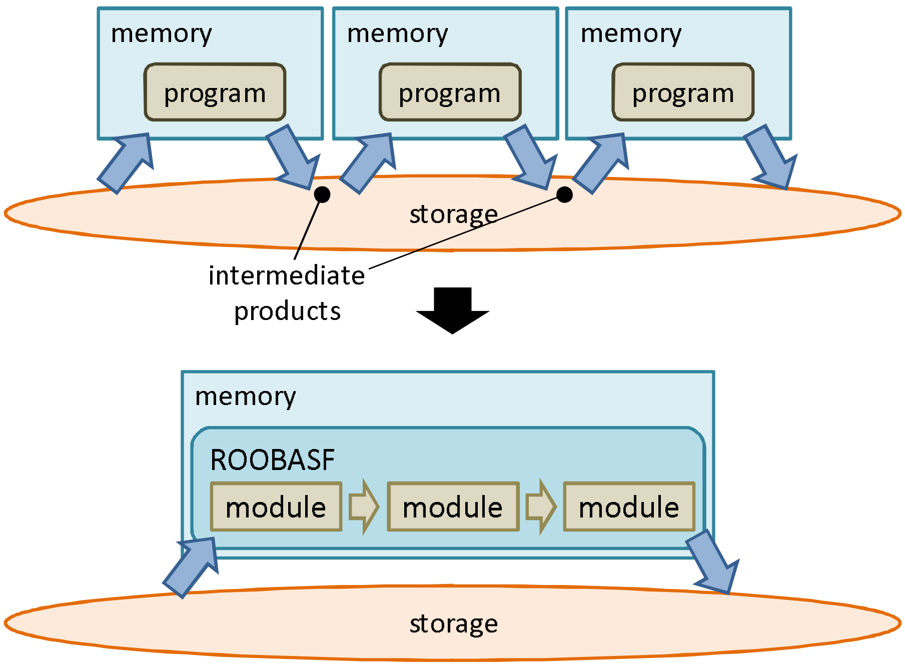


Figure 2.9: ROOBASF passing data on memory

a new framework natively supporting parallel computing that controls analysis workflow, as is described in the next chapter.

Flat-making pipeline

This pipeline is started on demand. It makes “flat” images, images of a pseudo-flat light source, indicating pixel sensitivity patterns of CCD sensors. The “flat” images are created by combining night/twilight sky images or dedicated dome flat images (images of the dome of the telescope uniformly lit.) The images are then used by the real-time analysis (Section 4.2, FLAT).

This pipeline is implemented by a C-shell script, which is invoked by the RCM controller.

Mosaicking pipeline

This pipeline is started on demand, and is implemented by a C-shell script. It superposes many CCD images from multiple exposures, searching for the best-fit positioning and determining flux scaling factors of respective images, and co-adds them into one mosaic image.

The job takes a large amount of calculations because it has to solve an optimization problem, to transform coordinates of the image (because the sky is curved,) to oversample the many pixels, and to co-add them all. Currently, the mosaicking process is run on one machine only because the mosaicking programs (SCAMP [2] and SWARP [3]) used in the C-shell script do not support *distributed memory* parallel computing.

The prototype pipeline manages to handle Suprime-Cam’s data, but it cannot deal with Hyper Suprime-Cam’s tenfold amount of images. We have to develop a program which mosaics images in parallel on distributed-memory machines for efficient mosaicking, by using all computers. Here, the new framework described in the next chapter is helpful in modularizing the program to make an on-memory mosaicking pipeline.

Chapter 3

Parallel Framework

In the previous chapter, we explained the necessity of a framework which controls analysis workflows and which supports parallel computing for fast data processing. The fast computing is what high energy physics is good at. We are therefore collaborating with the Belle II experiment group at KEK, the High Energy Accelerator Research Organization. The Belle II group has been developing a parallel framework, ROOBASF, for the next generation B-factory experiment. Based on the Belle II analysis framework, we have developed a new one for HSC. In this chapter, we describe the new framework named PBASF.

3.1 ROOBASF

BASF, the Belle AnalysiS Framework, is the current analysis framework used in the Belle experiment in KEK. A new framework has been developed for the future Belle II experiment. One of its features is that it innovates the ROOT[4] library by CERN, and the framework is named ROOBASF. We choose ROOBASF as the framework used in the prototype of the HSC real-time analysis pipeline (Section 2.3.3).

ROOBASF controls "modules" and "paths" in analyses (Figure 3.1.) Modules are plugins each corresponding to a stage of analyses. Users create their own modules, or use existing ones in libraries. ROOBASF arranges the modules in a line, and makes the modules process incoming "events," or units of data to analyze, in order. The line is called a "path." At the end of the path in which modules are arranged, ROOBASF can pass the events to another path in accordance with conditions.

Figure 3.1 also illustrates the parallelization supported by ROOBASF. ROOBASF creates processes ("event processes") and pools them so as to assign the pooled processes with events randomly. Each "event process" analyzes the assigned event according to the "path" and sends the result to the "output server." The "event processes" are independent of each others and we cannot make them communicate. In high energy physics, each event is analyzed inde-

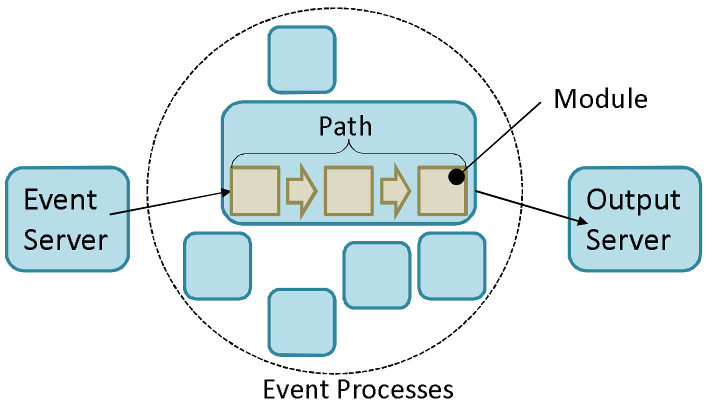


Figure 3.1: ROOBASF

pendently, and it is indeed unnecessary for the "event processes" to communicate with each others.

3.2 Parallel processing for HSC

In HSC, on the other hand, the requirement is somewhat different. In the HSC real-time analysis system, we are planning to assign processes with CCD images one by one. The process pooling method of ROOBASF does not fit here because CCD images in a shot or shots sometimes need analyzing together. A process has to communicate with others in such situations. We have to synchronize the parallel processes in order to have them communicate. Pooled processes are asynchronously and randomly assigned with events, and there is no way of its control.

We have developed another framework that synchronizes processes and allows them to communicate with each others. The basic concept of modules and paths are based on ROOBASF, and synchronization is actualized by the standard Message Passing Interface (MPI.)

3.3 Analysis framework with MPI

3.3.1 The Message Passing Interface

The Message Passing Interface (MPI) is a de-facto standard for parallel computing with distributed memory. It is an API library of multiprocessing — sending data, receiving data, and synchronizing, for example — standardized by the MPI Forum [5]. We can write portable parallelized programs using MPI because its definition is strict and sufficiently virtualized. There exist math libraries of third parties, and we can utilize them. There are many implementations of MPI such as Open MPI (LAM/MPI, previously) and MS MPI by Microsoft, and we employed MPICH2 [6].

An ordinary program with MPI is launched with multiple processes from the start. The program utilizes these parallel processes to accomplish its task. One of its features to note is that processes can be grouped. Groups are created dynamically and processes can belong to multiple groups. Each group has its "communicator." Using a communicator, processes in a group can together do group operations: broadcasting — making a datum in a process shared by all, gathering — having data in all processes gathered to one, reducing — calculating $\sum_{\text{processes}}$, $\prod_{\text{processes}}$, etc, etc. We utilize the communicators in our framework as is described in Section 3.3.5.

3.3.2 PBASF overview

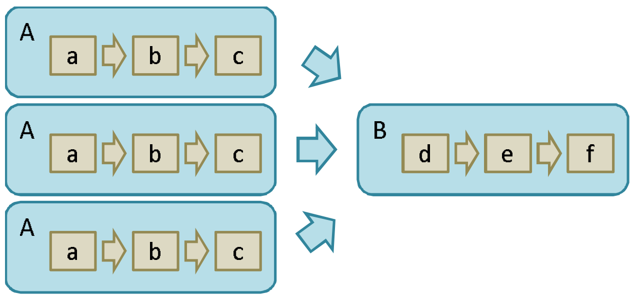


Figure 3.2: An example of par-paths in the framework

We have developed a framework that utilizes MPI. We named the framework "PBASF" standing for Parallel BASF. The name of ROOT is dropped because usage of the ROOT library is optional in this framework. It is written in C++, and has a Python interface. Users can write their own modules in C++ or Python. Python is a powerful and simple scripting language. Users can write Python scripts very easily and quickly. They do not have to use C++ as long as the modules written in Python are not too slow. With these two languages, users can efficiently develop analysis pipelines.

In PBASF, the concept of paths in ROOBASF is extended to be "parallel paths" or par-paths. An example of par-paths is shown in Figure 3.2. The squares 'a', 'b',..., 'f' denote analysis modules. They are prepared by users and each of them carries out one analysis step. Each of the sequences of 'a' to 'c' and 'd' to 'f' in rounded rectangles is executed in a process, and is called a traditional "path." The three rounded rectangles 'A' are executed in parallel, and the 'B' is executed asynchronously of 'A'. Analyzed data, called "events", are first processed by the three processes of 'A', then gathered, and finally processed by the single 'B'. This path consisting of four processes is named a "par-path."

PBASF builds up par-paths and controls flow of events in the par-paths. We describe paths and par-paths in detail below.

3.3.3 Paths

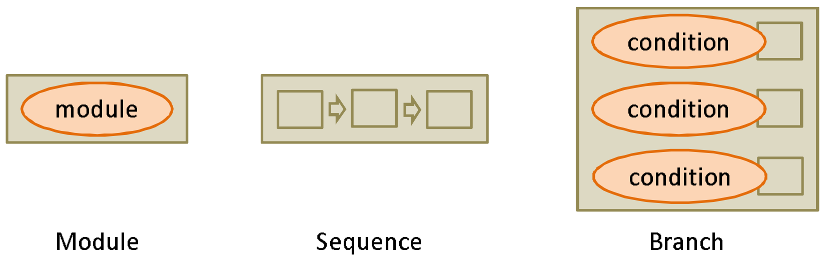


Figure 3.3: Three basic elements of paths

"Paths" define analysis flows *within processes*. They have nested structures and consist of the following three basic elements.

- Modules
  An analysis module written by users constitutes a path by itself.
- Sequences
  A sequence of multiple paths also becomes a path. An event, coming in the sequence, is passed to each path in the sequence in order.
- Branches
  A set of condition-path pairs becomes a path. An event, coming in the branch, is passed to one of the paths in the branch in accordance with condition.

We show an example of the nested structure of paths in Figure 3.4. The path in the example is equivalent to the procedure:

```
call mod.a
if cond.1
then
   call mod.b
else
   call mod.c
end
call mod.d
```

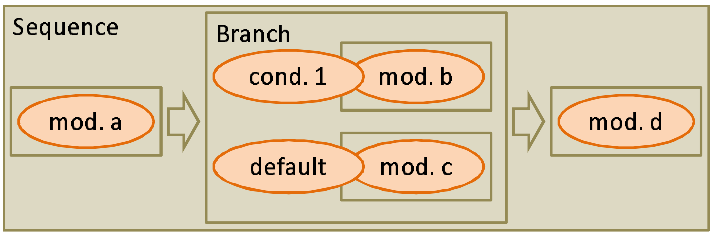


Figure 3.4: An example of nested path structures

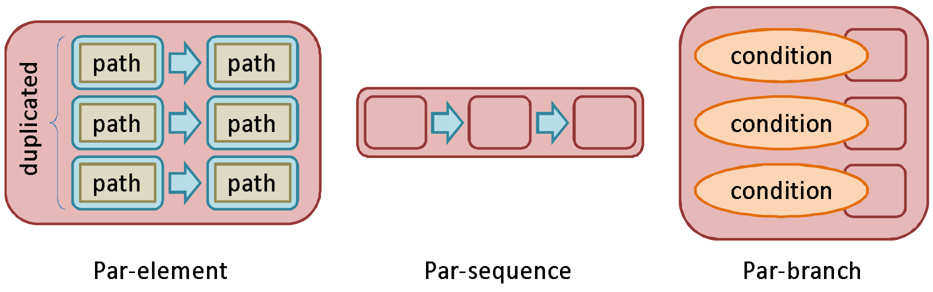


Figure 3.5: The basic components of par-paths

3.3.4 Par-paths

"Par-paths" being short for "parallel paths" define *inter-process* flows of analyses. They have nested structures similar to paths. Their components are as follow:

- Par-elements
  These are in the lowest layer of par-paths. A par-element is a sequence of *paths*. Each of the paths is assigned with one process. A par-element can be multiply duplicated. All the duplicated par-elements run in parallel. If a par-element contains two paths and is triply duplicated, then the total number of used processes is $2 \times 3 = 6$.

- Par-sequences
  A sequence of *par-paths* becomes a par-path (*cf.* Par-elements.)

- Par-branches
  A set of multiple "[condition, par-path]" pairs becomes a par-path. An event, coming in the par-branch, is passed to one of the par-paths in the par-branch according to condition.

The difference between a par-element and a par-sequence is that a par-element is a sequence of *paths* while a par-sequence is a sequence of *par-paths*. It also has to be noted that in a par-sequence events simultaneously analyzed by multiple processes are *gathered to one* at the end of each par-path before they are sent to the next. This is not the case with a par-element. In a par-element, events are not gathered until its end as seen in Figure 3.5.

3.3.5 MPI communicators

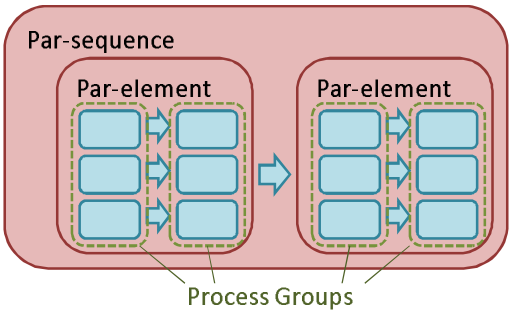


Figure 3.6: Process grouping

Processes are grouped, and the groups are assigned with communicators, as shown in Figure 3.6. That is to say, processes executing same paths are grouped. Processes in a group are re-numbered from 0 to one less than the number of processes. As a result, modules in a path (or, in a group) can do SPMD (Single Program Multiple Data) computing with the communicator just as if each of the modules were executing a usual MPI program.

3.4 Inside PBASF

3.4.1 Execution of paths

Paths have nested structures, and execution of them is done by just traversing the structures. There are several execution types in accordance with the data type read by an input module (*cf.* Section 3.6.2 and Section 3.6.3.) The behaviors of PBASF in response to the read types are divided into two types.

If it reads `PBASF_INST_EVENT`, the execution goes as the following:

- PBASF calls the top-level path's `event` member.
- Modules' `event` defined by users does their own task.
- Sequences' `event` member calls the `event` members of their elements in order.
- Branches' `event` member searches their contents for a path the condition associated with which returns true. Then call the path's `event` member.

Then the analysis modules contained in the whole path are executed properly.

When it reads an instruction other than `PBASF_INST_EVENT`, the execution goes as the following. We take `PBASF_INST_BEGIN_RUN` for instance.

- PBASF calls the top-level path's `begin_run` member.
- Modules' `begin_run` defined by users does their own task.
- Sequences' `begin_run` member calls the `begin_run` members of their elements in order.
- Branches' `begin_run` member calls the `begin_run` members of their contents. Conditions are not cared.

3.4.2 Assigning par-paths to processes

Par-paths have nested structures, and the assigning of it to processes is done just traversing the tree structure.

- PBASF passes P, the set of n processes, to the top-level par-path.
- Par-elements assign their paths with $a \times b$ processes from P where a is the number of the paths and b is the parallelization. Then the par-elements subtract the processes from P.

- Par-sequences pass P to par-paths in them in order.
- Par-branches do the same as Par-sequences.

That is all the job of par-paths. After the assignment, each process autonomously does the given task.

3.4.3 Tasks of processes

Each process, after the allocation of tasks, does its own task autonomously. The processing of events is simplified to the loop of the three procedures:

1. Receive an event to analyze.
2. Pass the event to the top-level path assigned to the process.
3. Send the event.
4. Go to 1.

Receiving and sending events are, however, a complicated procedure. If there exists no process from which to receive events, the receive procedure calls data-input module instead. If the process is at the end of a par-element, the send procedure has to gather results from all other parallel processes before sending them to the next processes (Figure 3.7.)

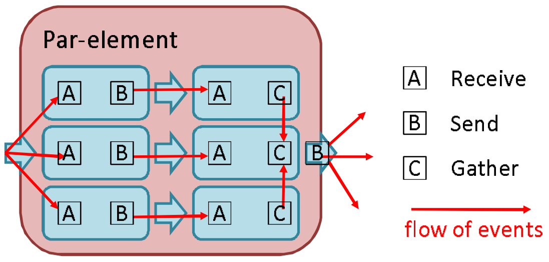


Figure 3.7: The flow of events

Events are thus relayed, but other instructions need extra cares. Instructions other than events are sent to all par-branches regardless of conditions. Naively, the instructions may take over a precedent event (Figure 3.8). In the figure, the "end_run" instruction is a marker dividing a set of events from another. It is not good the "end_run" taking over the preceding event.

The receiver in the rightmost process in Figure 3.8 therefore has to wait for all "end_run" to arrive before returning the instruction. The receiver then becomes like this:

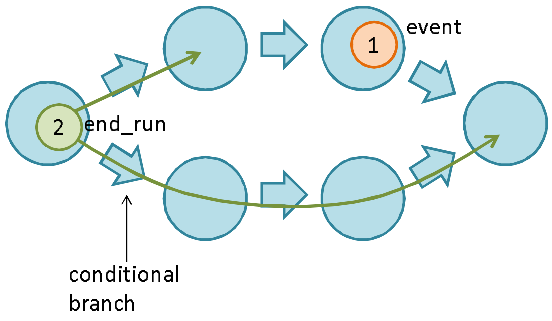


Figure 3.8: An end_run taking over an event

```
Procedure *receive* is:
begin
  Receive an instruction
  if the instruction is not an event
  then
    if all the branched instructions arrived
    then
      Send back acknowledges.
    else
      Call *receive* recursively.
    end
  end
  Return the instruction.
end
```

3.5 Building PBASF

PBASF is developed and tested on Cent OS 5, x86_64. We use only POSIX functions, and we expect it compiles in many other environments. We have made sure that it also works on Fedora Core 6, x86_64.

Other libraries required are:

- Python 2.4 or later
  PBASF has a Python user interface. Configurations are written in Python scripts. We describe the user interface in Section 3.7.

- MPICH2 [6]
  PBASF needs some MPI library. We have developed PBASF on MPICH2, but we have also made sure that PBASF compiles with Open MPI.

- BOOST C++ Library
  The BOOST library is required by PBASF. We are not sure of the lower

limit of its versions, but probably the version 1.38 or later works. The framework needs boost.mpi in particular, which is not built by default.

- boostmpi [7]
  boostmpi is a great Python wrapper around MPI developed by Doug Gregor. It is maintained and distributed by Andreas Klöckner.

3.6 Plugins

In this section, we describe what plugins are necessary for PBASF. All kinds of plugins can be written in C++ and Python. Writing modules in Python script is efficient in developing software, though these modules run more slowly than those in native code.

If users write a module in C++, they derive a new subclass from a base provided by PBASF. They then build a shared object (a dynamically linked library file on Linux systems) which exports a function like

```
using namespace pbasf;
extern "C"
Ptr<CPlugin<CModule> >
Foo_NewModule()
{
  return Gc(new CPlugin<CFoo>(
    Gc(new CFoo())
  ));
}
```

where `CFoo` is the class users want to export. `Ptr` is a smart pointer template, and `Gc`, standing for "garbage collection," is a template function that converts a raw pointer to a smart ptr. As seen in the code, the instance of `CFoo` is wrapped in a `CPlugin`, which is returned by the exported function. The name of the exported function is in the format: "the name of shared object file," an underscore, and "New***", where *** varies in accordance with the type of the plugin.

If users want to write a plugin in a Python script, they derive a class from an appropriate base class, and just instantiate it in configuration scripts for PBASF.

3.6.1 Events

Users define their own event class. The class can be defined either in C++ or in Python. In C++, users must derive it from `pbasf::CEvent`. In Python, users must derive it from `boostpbasf.CEvent`. If users define the event class in Python, they can access from C++ to its members via the boost::python library.

The abstract base class `pbasf::CEvent` or `boostpbasf.CEvent` does not have any member fields nor member functions. It is users' responsibility to design what to pack into an event.

An event class, written in C++, is not a plugin dynamically loaded by PBASF. Instead, it should be a shared object linked from all other plugins.

3.6.2 Analysis modules

Users prepare analysis modules in C++ or in Python. In C++, users derive it from `pbasf::CModule`. In Python, users derive it from `boostpbasf.CModule`. Users can override the following member functions.

- `init`
  The `init` member is called first of all. Modules initialize their states.

- `term`
  The `term` member is called at the end of all. Modules free the memories they have allocated, and close the files they have opened.

- `begin_run`
  The `begin_run` member is called if data-input modules (Section 3.6.3) return `PBASF_INST_BEGIN_RUN`. It is a mark indicating the start of a set of events with a certain apparatus configuration.

- `end_run`
  The `end_run` member is called if data-input modules (Section 3.6.3) return `PBASF_INST_END_RUN`. It is a mark indicating the end of a set of events with a certain apparatus configuration.

- `event`
  The `event` member is called if data-input modules (Section 3.6.3) return `PBASF_INST_EVENT`. The event to analyze is passed to this member as an argument.

3.6.3 Data inputs

Processes are assigned data-input objects if they do not have predecessors (*i.e.* other processes from which events are sent.) Data-input objects are instances of a user-defined class. The class must derive from `pbasf::CDataInput` or `boostpbasf::CDataIn`, and users should override its function-call operator.

The overridden function-call must return the two: an instruction code to the analysis modules, and an event object. The instruction is one in the following list. When the instruction is not `PBASF_INST_EVENT`, the returned event object should be NULL or None.

- `PBASF_INST_INIT`
  This instructs all modules to initialize themselves.

- `PBASF_INST_TERM`
  This instructs the whole par-paths to terminate analysis after all modules clean up their messes. A data-input module typically returns the instruction at the end of the whole data.

- `PBASF_INST_BEGIN_RUN`
  This tells all modules the start of a set of events with a certain apparatus configuration.

- `PBASF_INST_END_RUN`
  This tells all modules the end of a set of events with a certain apparatus configuration.

- `PBASF_INST_EVENT`
  This instructs all modules that an event is read, and that they analyze the event.

3.6.4 Conditions

Conditions are user-defined modules that are derived from `pbasf::CCondition` or `pbasf.CCondit`. They override function-call members, which return true or false according to arguments. Condition modules are used in path branches and par-path branches.

3.6.5 Forkers and joiners

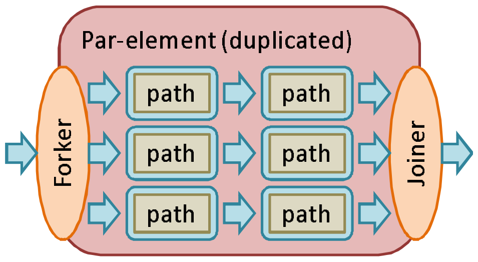


Figure 3.9: A forker and a joiner at the both ends of a par-element

Forkers divide an event to pieces at the beginning of duplicated par-elements, while joiners bond multiple events to one at the end of duplicated par-elements (Figure 3.9.) Users derive them from `pbasf::CForker` / `pbasf::CJoiner` in C++ or `pbasf::CForker` / `pbasf.CJoiner` in Python. They override their function-call operators that divide or bond events.

3.6.6 Serializers and deserializers

PBASF uses a user-defined serializer and a deserializer to de/serialize events in order to transmit them from process to process. Users derive them from `pbasf::CSerializer` / `pbasf::CDeserializer` in C++ or `pbasf::CSerial` / `pbasf.CDeserial` in Python to override their function-call operators. A typical implementation is to use boost.serialization or Python's pickle.

3.7 User interface

3.7.1 PBASF in Python scripts

PBASF provides a Python interface via boost.python, which is named "boostpbasf." Its entity is a shared object "boostpbasf.so" which can be imported into Python. It works as a usual python module. boostpbasf has the following member classes.

- `CEvent`
  `CEvent` is the base class of events. Users can derive a subclass from it. Note that Python analysis modules are given run time type information of the event object being analyzed.
- `CModule`, `CDataIn`, `CCondit`, `CForker`, `CJoiner`, `CSerial`, `CDeserial`
  These are the base classes of PBASF modules. Users can derive subclasses from them. Details are described it in Section 3.6.
- `CFrame`
  `CFrame` is the main console of the framework, PBASF. Users command PBASF via the instance of `CFrame` to load plugins, to build path and parpath graphs, and to assign processes to all asynchronous or parallel tasks in the graphs. We describe how to use it below. We use `fr` as an instance of `boostpbasf.CFrame` in the rest of sections in the chapter.

3.7.2 Loading plugins

In order to load an analysis module plugin, users call

```
fr.Module('ID', 'FileName')
```

where 'ID' is an identifier used in PBASF and 'FileName' is the name of the plugin without its extension. If they are to load a data-input module, they call `fr.DataIn('ID', 'FileName')`. They also call `Condit`, `Forker`, `Joiner`, `Serial` and `Deserial` accordingly.

Multiple instances of a plugin can be loaded, given their identifiers are different. If only one instance is necessary, they can call these functions with one argument like `fr.Module('ID')`. In this case, the file name is assumed to be the same as 'ID'.

3.7.3 Creating paths

To explain how to create paths, we first present, in Figure 3.10, the Python code to create the one in Figure 3.4.

```
fr = boostpbasf.CFrame()                 # create a PBASF console
fr.Module('mod.a', 'a')                  # load module 'a.so' as 'mod.a'
fr.Module('mod.b', 'b')                  # load module 'b.so' as 'mod.b'
fr.Module('mod.c', 'c')                  # load module 'c.so' as 'mod.c'
fr.Module('mod.d', 'd')                  # load module 'd.so' as 'mod.d'
fr.Condit('cond.1', '1')                 # load module '1.so' as 'cond.1'
                                         #
fr.Branch_Add('bra', 'mod.b', 'cond.1')  # create a branch named 'bra'
fr.Branch_Add('bra', 'mod.c')            # to which add 'mod.b' and 'mod.c'
                                         #
fr.Seq_Add('seq', 'mod.a')               # create a sequence named 'seq'
fr.Seq_Add('seq', 'bra')                 # to which add 'mod.a', 'bra'
fr.Seq_Add('seq', 'mod.d')               # and 'mod.d'
```

Figure 3.10: A Python code to build the path in Figure 3.4

Users call `fr.Seq_Add('ID',...)` and `fr.Branch_Add('ID',...)` to create path graphs. The first argument 'ID' is the identifier of the "sequence" or "branch" accordingly.

If `fr.Seq_Add('ID', 'ModID')` is called, the module 'ModID' is appended at the back of the sequence 'ID'. A sequence or branch with a certain identifier is created with no contents on the first use of the identifier.

`fr.Branch_Add('ID', 'SeqID', 'ConID')` adds the sequence 'SeqID' with condition 'ConID' to the branch 'ID'. If the condition is omitted, it treats the sequence as the default sequence. Paths have nested structures. Any of modules, sequences, and branches can be added to any sequences and branches.

We described above how to use shared object plugins written in C++. The usage of Python module is simpler. For example:

```
foo = CFoo()              # instanciate class CFoo(boostpbasf.CModule)
fr.Seq_Add('seq', foo)    # add it to 'seq'
```

3.7.4 Creating par-paths

Creating par-paths is similar to the case of paths. We present in Figure 3.11 an example Python code to create par-paths shown in Figure 3.12.

Users call `fr.Par_Add('ID', 'path')` to construct par-elements. The par-element with the 'ID' is created on the first call to `fr.Par_Add` with the ID. And paths are appended in order. The par-element can be duplicated to run in parallel with a call to `fr.Par_Parallel`. The call takes as its arguments the number of parallelism, a forker, and a joiner so as to split and gather events at the edges of the par-element.

```
(...)                                              # (Paths are already created.)
nulfork = boostpbasf.CForker()                     # Default forker
nuljoin = boostpbasf.CJoiner()                     # and joiner do nothing.
fr.Forker('fork', 'myfork')                        # Load ‘myfork.so’
fr.Joiner('join', 'myjoin')                        # and ‘myjoin.so’.
                                                   #
fr.Par_Add('par.P', 'path.a')                      # Create a par-element ‘par.P’
fr.Par_Add('par.P', 'path.b')                      # and add paths to it.
fr.Par_Parallel('par.P', 3, nulfork, 'join')       # Make ‘par.P’ parallel
                                                   # with a forker and a joiner.
fr.Par_Add('par.Q', 'path.c')                      # Create a par-element ‘par.Q’
fr.Par_Add('par.Q', 'path.d')                      # and add paths to it.
fr.Par_Parallel('par.Q', 2, 'fork', nuljoin)       # Make ‘par.Q’ parallel
                                                   # with a forker and a joiner.
fr.PSeq_Add('pseq', 'par.P')                       # Create a par-sequence and to
fr.PSeq_Add('pseq', 'par.Q')                       # which add ‘par.P’ and ‘par.Q’
```

Figure 3.11: A Python code to build the par-path in Figure 3.12

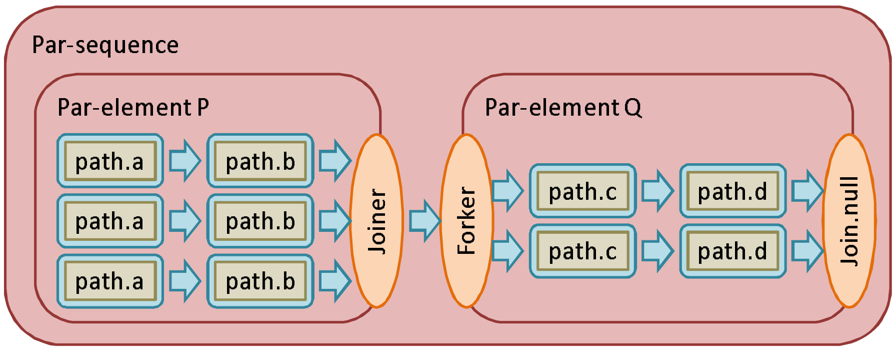


Figure 3.12: An example of a par-sequence

`fr.PSeq_Add('ID', 'par')` adds a par-path 'par' to the par-sequence 'ID'. The par-sequence with the ID is created on its first use. Par-sequences bond par-paths together.

`fr.PBranch_Add('ID', 'par', 'cond')` adds a par-path 'par' to the par-branch named 'ID'. Events are sent to 'par' on condition that the condition module 'cond' returns true. `PBASF_INST_BEGIN_RUN`, `PBASF_INST_END_RUN`, and other instructions except for `PBASF_INST_EVENT` are sent to all branches irrespective of the condition. It is guaranteed that these instructions, including `PBASF_INST_EVENT`, never get out of order on the way through branching and confluence.

3.7.5 Assigning and running tasks

After the creation of the top par-path, we split it by the following Python code.

```
(...)                                 # (Par-paths 'pseq' is already created.)
task = fr.PSeq('pseq').GetTask()      # Get the task of this process.
```

All MPI processes run the same Python script at the same time until this line. The 'GetTask()' then splits the total task and returns the assignment for each process.

```
(...)                              # 'task' is assigned already.
boostpbasf.RunRelay(               #
   task,                           #
   fr.DataIn ('MyDataIn'),         # Load 'MyDataIn.so' and use it.
   fr.Serial ('MySerial'),         # Load 'MySerial.so' and use it.
   fr.Deserial('MyDeserial'),      # Load 'MyDeserial.so' and use it.
   0                               # initial status code
)                                  #
```

Figure 3.13: A python code to start event relays

The preparation work has been done, and we are ready to have events processed in relay by the MPI processes. We call 'RunRelay()' and event relays start immediately (Figure 3.13.) The data-input module is attached to all those processes with no process from which to receive events. The relays continue until the data-input module returns `PBASF_INST_TERM`.

3.7.6 Launching scripts

We have described above how to write configuration scripts for PBASF. Now, the command line to launch the scripts is slightly different from that of ordinary Python scripts:

```
bash$ mpiexec -n 10 python config.py ↵
```

We have to prepend “mpiexec -n [the number of processes]” to the command line (in the case of MPICH2.) In the command line above, 10 processes are launched and they execute config.py where a configuration of PBASF is described.

Chapter 4

Pipeline Implementation

There exists a prototype of a real-time analysis system for Hyper Suprime-Cam (Section 2.3.3.) The prototype has been developed by NAOJ and is tested with data from Suprime-Cam. The prototype is currently based on a primary development version of ROOBASF. We ported the pipeline to PBASF.

In this chapter, we describe the implementation of the pipeline in PBASF. Its performance test is described in the next chapter.

4.1 Pipeline overview

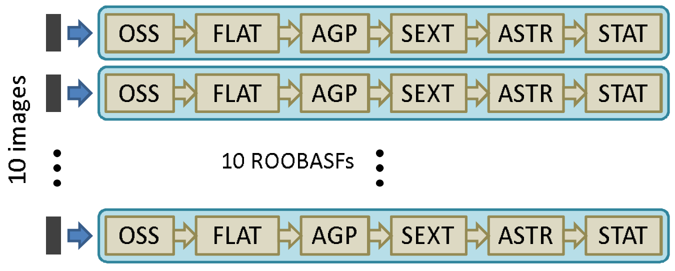


Figure 4.1: The original pipeline in the analysis system prototype

Figure 4.1 shows the current ROOBASF pipeline corresponding to '1a' in Figure 2.7. Suprime-Cam has 10 CCDs on the focal plane, and a shot produces 10 images at the same time. ROOBASF, however, did not have any parallelization capability (process pooling) when we started to develop the prototype. Therefore, 10 ROOBASF processes are launched in parallel to process 10 images coming from Suprime-Cam at a time. Because the 10 ROOBASFs are independent processes, there is no way for analysis modules in the pipeline to communicate with each others.

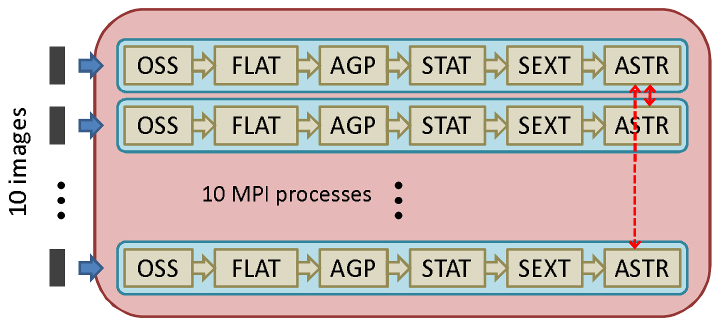


Figure 4.2: The new pipeline on PBASF

We show in Figure 4.2 the newly ported pipeline on PBASF. The total 10 processes are now wrapped in PBASF. The framework enables the processes to communicate with MPI, and the last module 'ASTR' now communicates with each others.

4.2 Modules

It is easy to make existing stand-alone C/C++ programs to be PBASF analysis modules, if they are small. We have only to change the name of the main function to some different one, and to call it from the `event` member function of the module class. We describe below each modules in the pipeline. All of those are ported from existing programs.

- OSS: Over-scanned region subtraction
  Charge in a pixel in an image can be considered to be proportional to the number of photons that enter into the pixel. However, the read-out signal height of the pixel is not linear to the change because of a zero-point value. To estimate the zero-point value, we apply "over-scanning". In Figure 4.3, we show how to read out a CCD sensor. The three continuous boxes in the figure represent pixels. The charges in the pixels are shifted and shifted so that the one amplifier reads them out one by one. If we shift the charges excessively ("over-scan,") we can get estimate the value corresponding to zero photons (Figure 4.3.) The over-scanned values are stored in an image together with ordinary pixel values, forming "over-scanned regions." This module subtracts the over-scanned values from the ordinary pixel values to adjust zero points.

- FLAT: Flat-fielding
  Non-uniformity of the pixel gains is corrected in this module. The non-uniformity is due to the non-uniform pixel sensitivities of CCD sensors and to intervening optical systems. This module divides each pixel values of

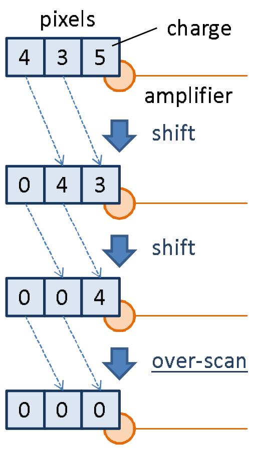


Figure 4.3: CCD over-scanning

the analyzed image by that of a prepared image of a “flat” light source to adjust the sensitivities of pixels. The “flat light” source is acquired from the twilight sky, the dome ceiling lit by artificial light, or the background of the night sky with stars and galaxies eliminated. The source would, ideally, produce uniform pixel values. The actual fluctuation of the pixel values is thus considered to be that of pixel sensitivities.

- AGP: Auto-guide probe masking
  Suprime-Cam has an auto-guide probe so that the telescope can track an object in accordance with the rotation of the celestial sphere. But this probe makes a shadow on the CCD. This module masks the shadow region on the image.

- SEXT: SExtractor
  SExtractor is a famous program by Emmanuel Bertin that builds a catalog of objects from an astronomical image [8]. It is originally a stand-alone executable, but we patched it to be a dynamic linked library. The patched sextractor can deal with memory files for its inputs and outputs, omitting disk I/O.

- ASTR: Astrometry
  This module calls SCAMP, a program written by Emmanuel Bertin, that reads SExtractor’s catalogs and computes astrometric and photometric solutions [2]. SCAMP is also patched just like SExtractor. The ASTR module has a change from the one in the original pipeline. As expressed by red arrows in Figure 4.2, the instances of this module communicate with each others. With the communication, catalogs from SExtractor are gathered to one process, and the one process calls SCAMP. SCAMP ac-

cesses a server in the Internet to get a reference catalog. If many SCAMPs access to the server at a time, like the original pipeline, the server may not be able to handle all the requests. In addition, multiple catalogs input into SCAMP, better astrometric / photometric solutions can be expected. The catalogs are small in size unlike raster images, and the transmission time does not matter.

- STAT: Statistics
  This module extracts parameters indicating the quality of the analyzed image. This module calls SExtractor again with another set of input parameters than SEXT.

It requires special cares to make a PBASF module from a large program. The program being an independent executable, static or global variables are initialized automatically at the start, and allocated memories and opened file handles are freed and closed at the end. In contrast, PBASF does not prepare a process per call to the analysis module. we therefore have to take care so that the global or static variables initialized properly, that the module does not have a memory leak, and that all the handles are closed before the execution is returned to the caller.

Chapter 5

Performance Test

We described a pipeline implementation in PBASF in the previous chapter. In this chapter, we describe the performance tests of the pipeline. We installed PBASF on three Dell PowerEdges (OS: Cent OS 5 x86_64 / CPU: 1.86GHz 4cores) connected to each others with Gigabit-Ethernet. We measured execution time of the pipeline with several numbers of parallel processes to demonstrate the effect of the parallelization.

First, we describe a performance test in a configuration where all storage was shared with NFS. We found problems on the NFS use in large data analyses. Second, we describe a setup with the use of NFS minimalized and its performance. Then we scale the obtained result for SC and HSC cases.

5.1 Problems on the use of NFS

We first had the three machine share a hard disk with NFS, the Network File System.

5.1.1 Experiment setting

We connected the three computers with Gigabit-Ethernet as shown in Figure 5.1. One hard disk was shared by the three computers with NFS.

We prepared 130 images produced by Suprime-Cam. There are 10 CCDs on SC, and 130 images correspond to 13 exposures. The size of each image was 2k $\times$ 4k pixels. With the pixel depth being 16 bits, the image size corresponds to 16 MBytes. We then put these images into the pipeline shown in Figure 4.2. We prepared the following options in the pipeline configuration.

- The number of parallel processes
  We tested the pipeline with the number of parallel processes 1, 3, 6, and 9. In the case of one process, the pipeline was executed on one machine only, and images were analyzed one by one. If the number of processes was $n \times 3$ each of the three machines was assigned with n processes, and $n \times 3$

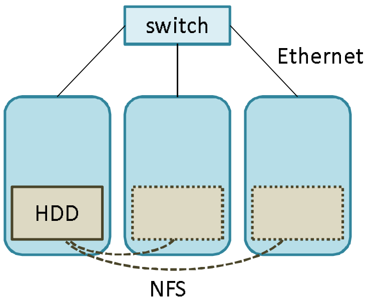


Figure 5.1: Experiment setting with NFS

images were analyzed in parallel. The total numbers of images analyzed varied slightly in accordance with the number of processes. They were 130 ($= 130 \times 1$), 129 ($= 43 \times 3$), 126 (21×6), and 126 (14×9), corresponding to 1, 3, 6, and 9 processes.

- Option to save intermediate products
  The size of each intermediate image was 32 MBytes. It was twice that of the initial image, because pixels of intermediate and output images were of 32-bit floating-point number type so that we could reduce computing errors. Four intermediate images were created per input image, and resulting $4 \times 32 = 128$-MByte intermediate images to be written on the hard disk. We therefore made an option whether or not to save the intermediate products.

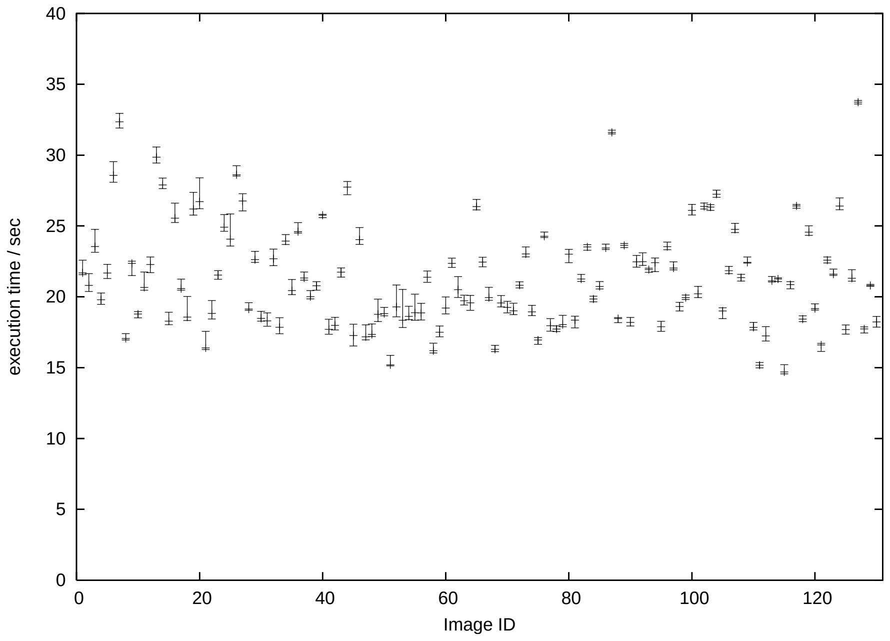
5.1.2 Requirement

The requirement for the execution time of the pipeline was 20-25 seconds per exposure to catch up with the quickest observation supposed for Hyper Suprime-Cam. One exposure of Hyper Suprime-Cam will produce ~ 100 images, hence 100 images should be digested in 20–25 seconds.

We tested the pipeline with image sets from SC, which produces 10 images per exposure. Ten images should ideally have been analyzed with 10 processes in parallel. We however measured the execution time for $3 \times n$ processes, because we had only three machines for the test, as described in the previous section.

5.1.3 Result

Figure 5.2 shows the measured execution time to analyze images one by one, *i.e.* using one process only. We repeated the same analyses 8 times, and calculated their medians. In the upper one, we stored all intermediate products in the hard disk, while in the lower one we did not. The two plots were similar. The analyses with one process did not use data transfer on the Ethernet because



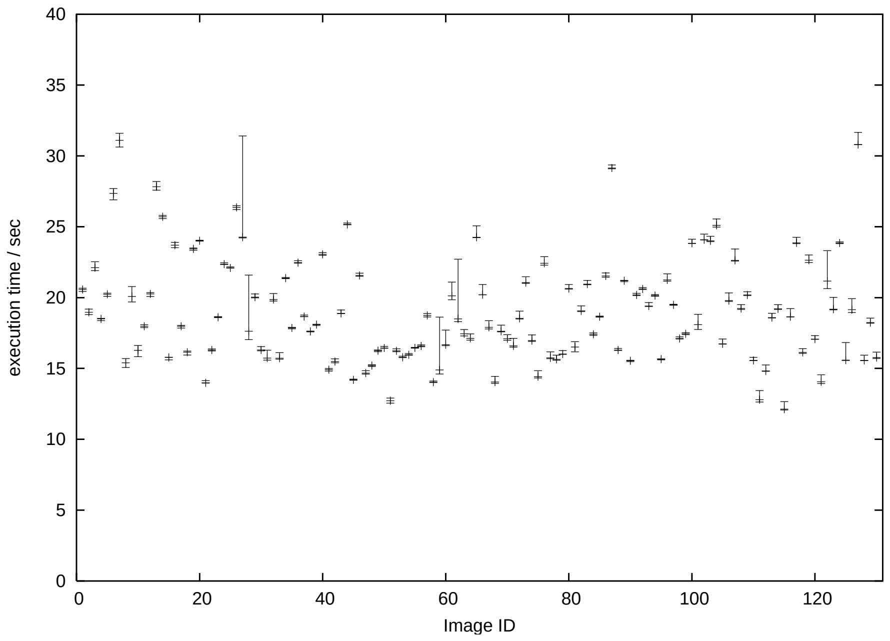


Figure 5.2: Execution time to analyze each image with NFS, one by one. The upper one shows the result with all intermediate products saved. The lower one shows the result with only the last product saved. Bars represent quartiles.

the process ran only locally on the computer with the actual storage, We thus conclude that data transfer time did not matter when data were not transferred via Ethernet.

The execution time was distributed between $\sim$ 15 seconds and $\sim$ 30 seconds. We have to note that there were a few cases ($\sim$ 0.3%) where analyses failed midway (probably on SExtractor or SCAMP) and the analysis time was > 50 seconds. Some of the cases took more than 200 seconds.

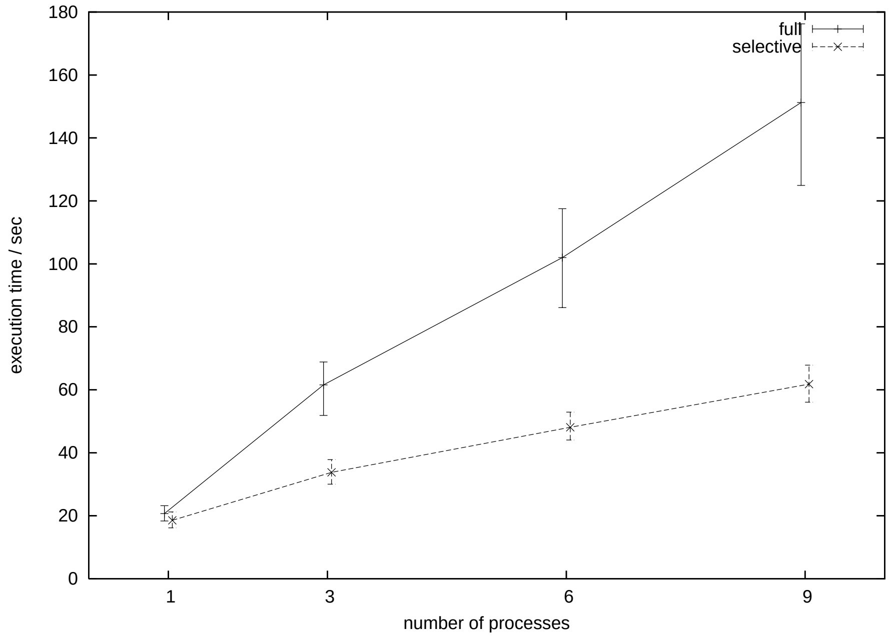


Figure 5.3: Execution time – number of processes relation with NFS. 'full' are the data with all intermediate products saved, while 'selective' are the data with only the last product saved. Bars represent quartiles.

We next show the relation between the execution time and the number of processes in Figure 5.3. In the figure, we see that, in the case of fully saving the intermediate products, the execution time was proportional to the number of processes. If the number of processes was three, the three processes were distributed to the three computers, and each machine executed only one process. The execution time should not have increased from the case of one process. As it happened, the execution time was increased as the process number, and we guessed that NFS transfer time had been dominant to mess up the advantage of parallelization.

In fact, the execution time significantly decreased when we did not save intermediate products of 128 MBytes per input image, as shown in the figure. We show the data traffic breakout in Table 5.1. In the table, the "input" images

Table 5.1: Data traffic on the Ethernet

	full saving	selective saving
input	16 MBytes	16 MBytes
flat reference	32 MBytes	32 MBytes
intermediate products	128 MBytes	—
last product	32 MBytes	32 MBytes
total	208 MBytes	80 MBytes

are those to be analyzed and the "flat reference" images are those of pseudo-flat light source used by the 'FLAT' module in Figure 4.2. These two were prepared on the NFS storage. The "Intermediate products" and the "last product" images were output into the hard disk via NFS. In the selective saving case, the data traffic on the Ethernet was 80MBytes/208MBytes $\sim$ 40% reduced.

Nevertheless, the intermediate products *should* be saved on the disk so that we can check them if the last products seem wrong, or the analyses seem to have failed on their way. The intermediate products saved on local hard disks, as we concluded comparing the two plots in Figure 5.2, had only a small effect on the execution time. We therefore changed the system configuration to use local disks proactively, and examined the analysis performance again, as described in the next section.

5.2 The effect of parallelization

This time, we prepared input data set on all local disks and saved all products onto respective local disks (Figure 5.4.) Flat reference images were also prepared on every local disk. Executables and libraries were still shared with NFS. Then we performed the same tests as in the previous section.

5.2.1 Analysis time of each image

We show in Figure 5.5 the measured execution time to analyze images one by one with only one process (black points.) We repeated the same analyses 9 times, and calculated their medians. The analyses with one process did not make any data transfer on the Ethernet just as in the case of Figure 5.2. The analysis time depends on respective input images, and we used the same data set as in Figure 5.2. The analysis time in Figure 5.5 was thus expected to be the same as in Figure 5.2. For comparison, we overlay onto Figure 5.5 the measured execution time shown in Figure 5.2 (red points.) We obtain good agreements between them, as shown in the figure.

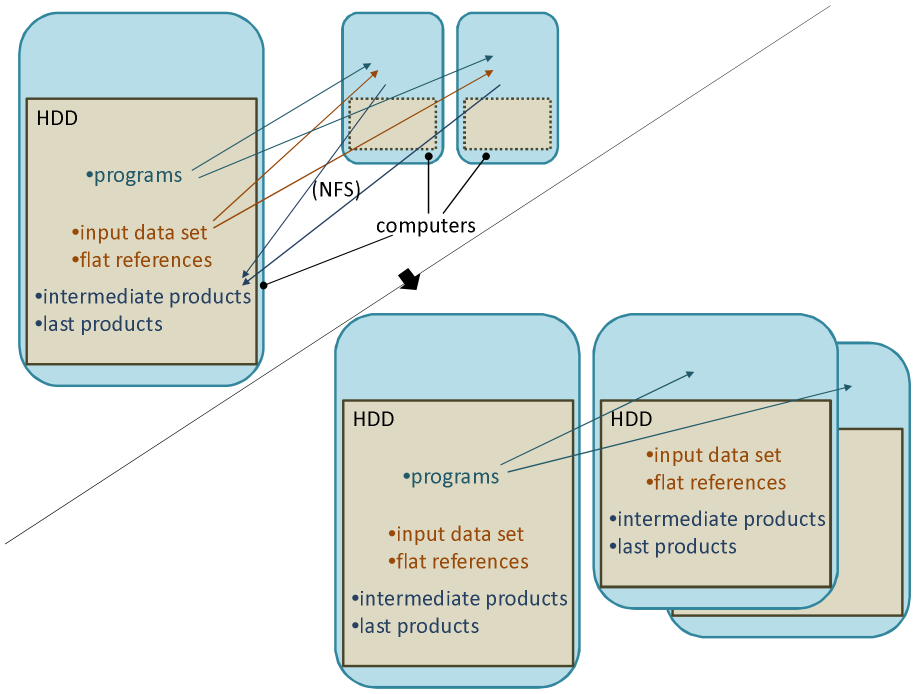
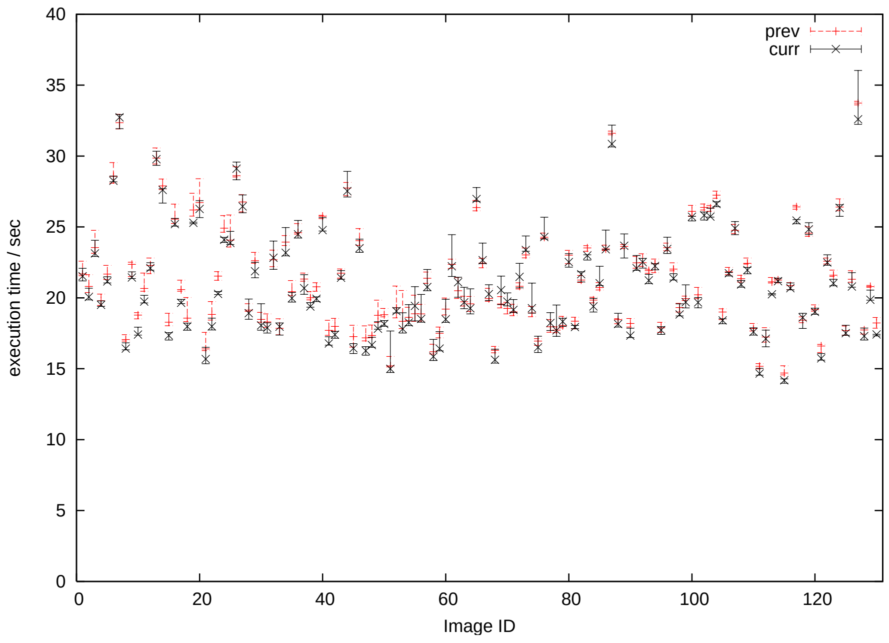


Figure 5.4: The change of the testing configuration. The upper-left picture illustrates the previous setup, while the bottom-right the new configuration.



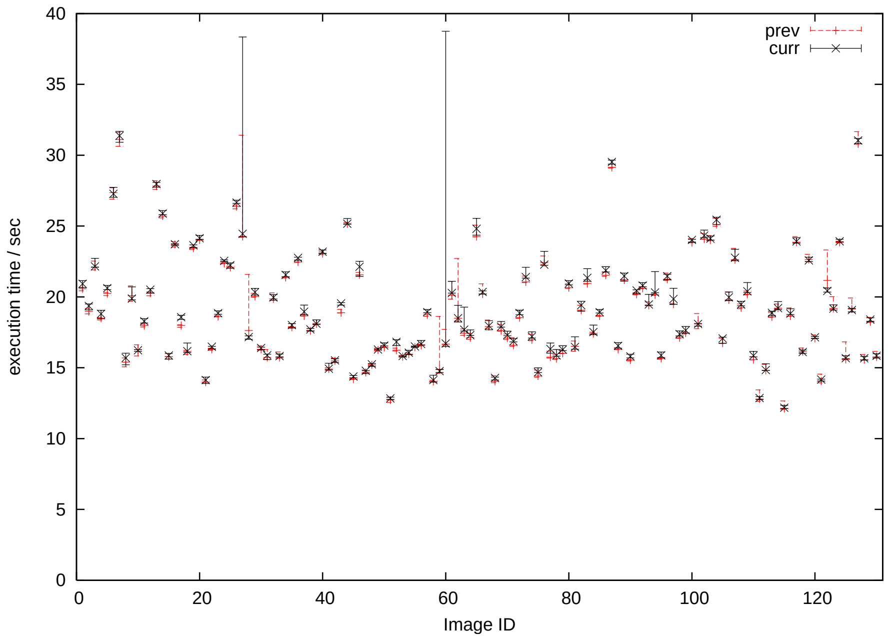


Figure 5.5: Execution time to analyze each image without NFS, one by one. The upper one shows the result with all intermediate products saved. The lower one shows the result with only the last product saved. Bars represent quartiles. The red points are copied from Figure 5.2 for comparison.

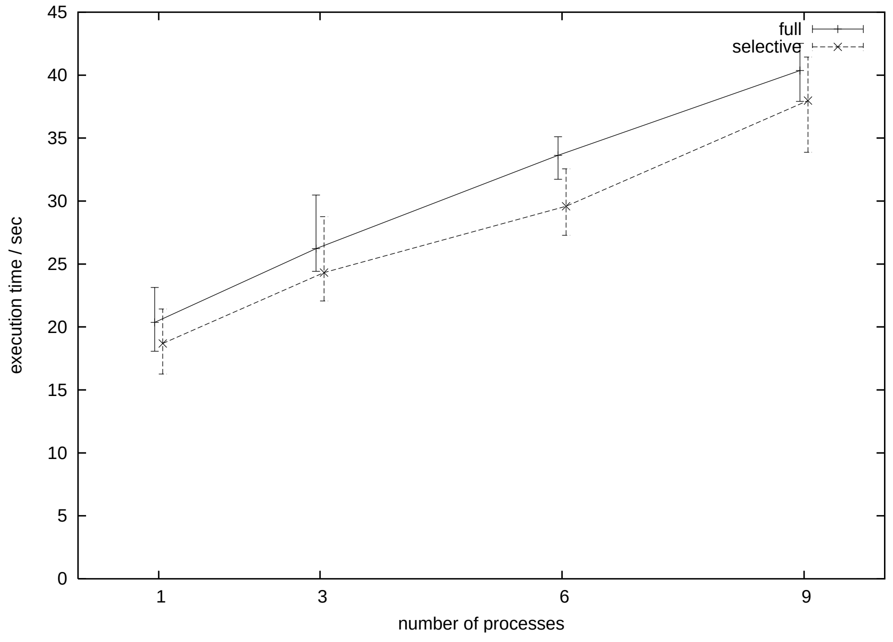


Figure 5.6: Execution time – number of processes relation without NFS. 'full' are the data with all intermediate products saved, while 'selective' are the data with only the last product saved. Bars represent quartiles.

Table 5.2: Comparison between the expected and the measured time. "(t_i)" in the table corresponds to the case of one process in Figure 5.6. "measured" is from the case of three process in Figure 5.6. The format of values is ${q_2}^{q_3-q_2}_{q_1-q_2}$ where q_i represents i-th quartile.

	(t_i)	(m_i)	measured
full saving	20^{+3}_{-2} sec	24^{+2}_{-2} sec	26^{+4}_{-2} sec
selective saving	19^{+3}_{-2} sec	22^{+2}_{-2} sec	24^{+4}_{-2} sec

5.2.2 Parallelization

Figure 5.6 shows the relation between the measured analysis execution time and the number of processes. Here, the execution time gradually increased as the number of processes increased. But the increase was much more moderate than in Figure 5.3.

We see that the execution time increased again from the case of one process to the case of three processes. As we mentioned before, the two values should be the same because each of the processes monopolized one of the machines when the number of processes was 3. This time, unlike Figure 5.3, the effect of NFS was minimized. Then one of the causes of the increase was expected to be the synchronization at 'ASTR' module for inter-process communication. The fastest process had to wait for the slowest one to come up. As a result, the execution time in the case of three processes was actually the maximum one of the three parallel processes. We describe its demonstration below.

We randomly generated t_i, analysis time of images, that follows the distribution shown in Figure 5.5. Then we made an array (m_i) from (t_i) using the following relation:

$$m_i = \max(t_{3i}, t_{3i+1}, t_{3i+2}) \quad (i \geq 0.)$$

The median of (m_i) was expected to be some larger than that of (t_i) because taking the maximum value in a 3-tuple picks up the fluctuation of its elements. We show the result in Table 5.2. We see in the table that the median of (m_i) was actually larger than (t_i) and it accounts for 2/3 of the increase of the analysis time from the case of one process to the case of three.

Figure 5.6 also shows that the increase of analysis time from 3 processes to 9 processes was linear with an offset. When the number of total processes was $n \times 3$, each machine was assigned with n processes. Each machine having 4 CPU cores the n $(\in \{1, 2, 3\})$ processes run almost in parallel. However, some of the analysis modules (SExtractor and SCAMP) in the pipeline were multi-threaded. They exhausted all the four cores and they could not run in parallel, causing the increase in the execution time. The last module 'ASTR' in the pipeline used group operations. The hard disk I/O operations were only serial. These also affected to the increase.

Table 5.3: Extrapolated analysis time, or median values of maxima of N-tuples of execution time. The format of values is ${q_2}^{q_3-q_2}_{q_1-q_2}$ where q_i represents i-th quartile.

(t_i)	$(m_i^{(10)})$	$(m_i^{(100)})$
20^{+3}_{-2} sec	27^{+4}_{-2} sec	41^{+26}_{-8} sec

5.2.3 Extrapolation of the measured time

We extrapolate the measured execution time in Figure 5.6 to estimate the execution time for 10 and 100 images analyzed in parallel, corresponding to SC and HSC respectively.

The requirement on the analysis execution time says that the analysis should be done in 20-25 sec per exposure. We assume the same machine performance as we used in this thesis. Then we should analyze CCD images in an exposure all in parallel, and should not assign more than one process to a machine, judging from Figure 5.6. We thus assumed that the number of the machines to be the same numbers of images in an exposure (10 in SC and 100 in HSC,) and that each of the machines is assigned with a process.

We then estimated the effect of synchronization of processes analyzing 10 or 100 images in an exposure, just as described in the previous section. We generated an array of analysis time of images, (t_i), that follows the distribution in Figure 5.5. We then made arrays $(m_i^{(N)})$:

$$m_i^{(N)} = \max_{Ni \le j < N(i+1)} t_j .$$

We assumed all images would be saved. We then obtained the median values of $(m_i^{(N)})$ as shown in Table 5.3.

Table 5.3 says that even if other effects are ignored, the synchronization of the processes alone will ruin the performance. The cause is the error rate not being small enough. As we noted above, there were a few cases where the analyses failed and the execution time became ~200 seconds. The probability of either of the parallel processes bumping into errors becomes larger with the number of processes. The probability was not prominent with 10 images in an exposure (SC), $(m_i^{(10)})$ being 27^{+4}_{-2} sec. However in the case of HSC, 100 processes running in parallel, the effect became nonnegligible. Hence, one of the issues to be improved is the reduction the error rate of analysis modules. Redesigning of the analysis and parameter tuning is currently underway.

Chapter 6

Conclusion

We have developed a parallel analysis framework, PBASF. The framework is derived from ROOBASF, which is being developed for the Belle II experiment. We introduced enhanced parallelism for HSC analyses in PBASF. PBASF utilizes MPI to manage many processes on many computers. It also uses Python in configuration and in writing modules, for efficient software development.

We ported into PBASF the pipeline used in the prototype of the on-line analysis system for HSC. While ten independent processes of the pipeline were originally run in parallel, the processes are controlled by PBASF in the ported pipeline.

The pipeline is required to run in less than 25 sec per exposure. Based on the performance test we did in a small prototype system, we scaled the results of the test and estimated the execution time to be at least a median of $\sim$40 sec per HSC exposure, even if 100 computers were used. We need more robust analysis modules to reduce the error rate responsible for the decrease in the performance of the parallel analysis pipeline. We expect that the problem resides in our usage of the analysis modules of SExtractor and SCAMP, and we are currently tuning their parameters. Though we expect that analysis failure occurred in these analysis modules, we have difficulty in pinpointing which module in what reason fails, since PBASF currently lacks a logger. Hence we need a logger in PBASF. We will implement the logger in PBASF, investigate the analysis error in detail, and improve the configuration of the real-time pipeline in order to achieve the requirement of 25 sec per exposure for the HSC analysis time.

In addition, PBASF offers feedback through the Python usage to ROOBASF for the Belle II experiment. The powerful scripting language allows for efficient software development in ROOBASF, while it does not interfere with existing native codes, or reduce their execution efficiency.

Appendix A

CCD Readout Clock Generator

In this chapter, we describe a signal generator developed for the readout hardware of CCDs on Hyper Suprime-Cam. [12] also refers to the signal generator, and we explain its details here.

A.1 Overview of the CCD readout

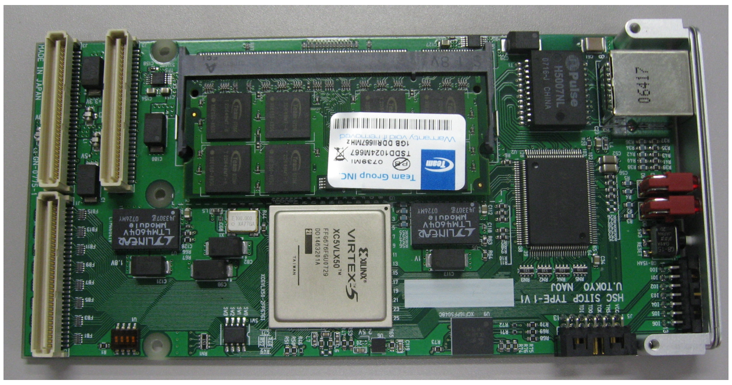


Figure A.1: GESiCA

Back-end digital circuits are being developed to control front-end analog circuits that read the CCDs on HSC. Among the back-end modules is a small board called "GESiCA" (Figure A.1) [13]. This board includes main functions necessary for the readout. The functions are as following.

GESiCA is connected to a computer with Gigabit-Ethernet. GESiCA generates readout signals (or, clocks) for the CCDs when it receives a command from the computer to read out CCDs. At the same time, it generates control signals for the analog readout circuits, and signals conversion timings for the ADCs. When the digitized image data come out from the ADCs, they are first stored in a buffer of DDR2-SDRAM and sent back to the computer via TCP. These functions are all implemented on the Xilinx Virtex-5, a Field Programmable Gate Array (FPGA). The circuits are written in HDL (Hardware Description Language.)

Among the functions of GESiCA, signal generation must be devised. CCD readout clocks are complicated parallel signals. In addition, the signals need altering according to observations. In fast observations the clocks should be fast. In slow observations the clocks should be slow so that the readout noise will be reduced. The change of CCD clocks is not independent of other signals. The front-end analog readout circuits require synchronization with the CCDs' output rate, and their controlling signals have to be changed accordingly. These things taken into consideration, GESiCA's output signals should not be hard-coded. The circuit in the FPGA of GESiCA therefore contains a module to generate these signals dynamically defined by users.

We call the module a "CCD readout clock generator." The module's output is nonetheless not confined to CCD clocks but the module generates the whole signals as well, controlling the front-end analog circuits.

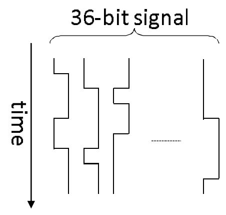
A.2 Architecture

The CCD readout clock generator is written in Verilog HDL, and works on Xilinx Virtex-5 FPGA. The clock generator is designed to be driven by a 100 MHz system clock. The clock should be as accurate as possible in order for a low jitter output signal.

The clock generator generates arbitrary 36-bit parallel signals, in which 32 bits are for essential use and the rest 4 are for future use. What users prepare are the following two: waveform fragments and sequencer programs. Then, a sequencer arranges the waveform fragments in accordance with the programs. Waveforms and programs are separately stored in block memories (RAM) inside the FPGA. The clock generator does not need external memory modules.

A.2.1 Waveform data

A pattern memory in the clock generator stores waveform data. Figure A.2 is an example of the waveform data. The 36-bit parallel signal in the left side is represented by a sequence of tuples of a bit pattern and a run-length shown in the right side. Each bit pattern is elongated for the accompanying run-length. The unit of the run-length is the clock given to the clock generator, which is expected to be 100 MHz. Each bit pattern in the table persists for $120/100\,\text{MHz} = 1200\,\text{ns}$. The shortest duration time of a bit pattern is 10 ns when its run-length is 1. The clock generator can thus output up to 50 MHz signals.



bit pattern (36 bit)	run-length (18 bit)
$001\cdots0_{(2)}$	120
$101\cdots0_{(2)}$	120
$110\cdots0_{(2)}$	120
$111\cdots0_{(2)}$	120
$011\cdots1_{(2)}$	120
$010\cdots1_{(2)}$	120
$100\cdots1_{(2)}$	120
$110\cdots1_{(2)}$	120
$110\cdots0_{(2)}$	120

Figure A.2: Waveform data

Table A.1: Sequencer's operation codes

field (bits)	8	8	8	8	8	8	8	8	8
mnemonic	verb	argument 1				argument 2			
end	0	0							
nop	0	# of clocks				0			
lstart	1	ID	0			# of loops			
lcont	2	ID	ID	ID	0	0			
ccdope	3	address		length		# of iteration			

With run-lengths, waveform data are compressed fairly efficiently because necessary signals seldom contain waveforms as high as 50 MHz. Hence, the clock generator can deal with waveforms long but finely tuned in tens of nanoseconds.

Multiple waveform fragments are stored in the pattern memory just continuously. No boundary information is stored along with the fragments.

A.2.2 Program data

Program data are stored in a program memory in the clock generator. The programs describe how to sequence waveform fragments stored in the pattern memory.

All operation codes are listed in Table A.1.

- `end`
  The opcode `0x00_00000000_00000000` stands for the end of a procedure. The sequencer stops when it comes into this opcode.

- `nop` (no operation)
  The opcode `0x00_XXXXXXXX_00000000` instructs the sequencer to stall for `XXXXXXXX` clocks (or, `XXXXXXXX` $\times$ 10 ns.) The value must not be 0 because it would be the `end` instruction.

- `lstart` (loop start)
  The opcode `0x01_NN000000_XXXXXXXX` instructs the sequencer to store the loop count `XXXXXXXX` in the `NN`-th count register together with the instruction pointer. `NN` can be 1, 2, or 3.

- `lcont` (loop continue)
  The opcode `0x02_NN000000_00000000` instructs the sequencer to decrement the `NN`-th count register, and to jump to the stored instruction address if the count is non-zero. Continuous opcodes `0x02_LL000000_00000000`, `0x02_MM000000_00000000`, and `0x02_NN000000_00000000` can be united to be `0x02_LLMMNN00_00000000` where `LL`, `MM`, and `NN` are either of 1, 2, or 3.

- `ccdope` (ccd operation)
  The opcode `0x03_AAAALLLL_IIIIIII` represents `IIIIIII` iterations of a waveform fragment that begins from address `AAAA` in the pattern memory, and that is `LLLL` long.

line	field (bits)				comment
	(8)	(16)	(16)	(32)	
1	3	10	20	100	Iterate 100 times the wf. from 10 to 10+20.
2	3	30	12	130	Iterate 130 times the wf. from 30 to 30+12.
3	0	0	0	0	End.

Figure A.3: An example of sequencer programs

A sample program is shown in Figure A.3. The program first iterates the waveform stored from address 10 to 10 + 20 (excluding the right edge) in the pattern memory, 100 times. Then the program continues to the waveform stored from address 30 to 30 + 12 in the pattern memory. After iterating the 2nd waveform 130 times, the program ends.

On the changing of waveform fragments, the sequencer does not make a gap. The output waveform is always a continuous sequence of fragments, except for loops. Here, we distinguish the term “loop” and “iteration.” Loops are represented by the operation code `lstart` and `lcont`. Iterations are represented by the argument of `ccdope` opcode.

The clock generator tries its best not to make a gap even in the case of loops, but gaps may occur if:

- `ccdope` before `lstart` is only 10 ns long. The reason is that it takes 10 ns for the sequencer to interpret `lstart`. Iterating a 10-ns fragment two or more times avoids a gap.

- `ccdope` before `lcont` is 60 ns long or less. This is from the fact that it takes 60 ns for the sequencer to make a jump.

The opcode for loops are given a twist. The sequencer allows up to triple loops, and the three loops have IDs 1, 2, and 3 respectively. If we naively require 3 lines at the end of a triple loop:

```
...
    loop3_continue
  loop2_continue
loop1_continue
```

then each line will consume 10 ns if a jump is not taken. It will take $60 + 20$ ns for the outmost loop to make a jump, undesirably.

Therefore, we made the loop continues combinable. The three loop continues above can be expressed in just one opcode, `0x02_03020100_00000000`. This opcode is executed all at once, and the sequencer can jump to any of the corresponding `lstart` in only 60 ns.

A.3 Inside the clock generator

The clock generator has two major sub-modules, the "pattern generator" and the "pattern sequencer." The pattern generator generates the waveform patterns (or, fragments) in the pattern memory, obeying the pattern sequencer. The pattern sequencer interprets instructions in the program memory and give commands to "pattern generator." In this section, we describe the two sub-modules.

A.3.1 Pattern generator

The pattern generator has the pattern memory containing (36 + 18)-bit wide waveform data as illustrated in Figure A.2. The pattern generator is given an *address*, a *length*, and an *iteration* value as its arguments. Then, the pattern generator generates a waveform fragment in the *length* stored from the *address* in the pattern memory. The generation is iterated as many times as the given *iteration* value. Any waveform in any length can be iterated any number of times with no stalling.

The inner structure is shown in Figure A.4. The address generator generates a read-address when the "renew address" signal comes. The read-addresses change as in the following pseudo-program:

```
for($i = 0$; $i <$ iteration; $++i$){
  for($a =$ address; $a <$ address + length; $++a$){
    yield $a$;
  }
}
```

At the beginning of the address generation, it stores given arguments to its registers and sets the "renew input" signal to instruct an external module (probably the pattern sequencer) to renew the arguments.

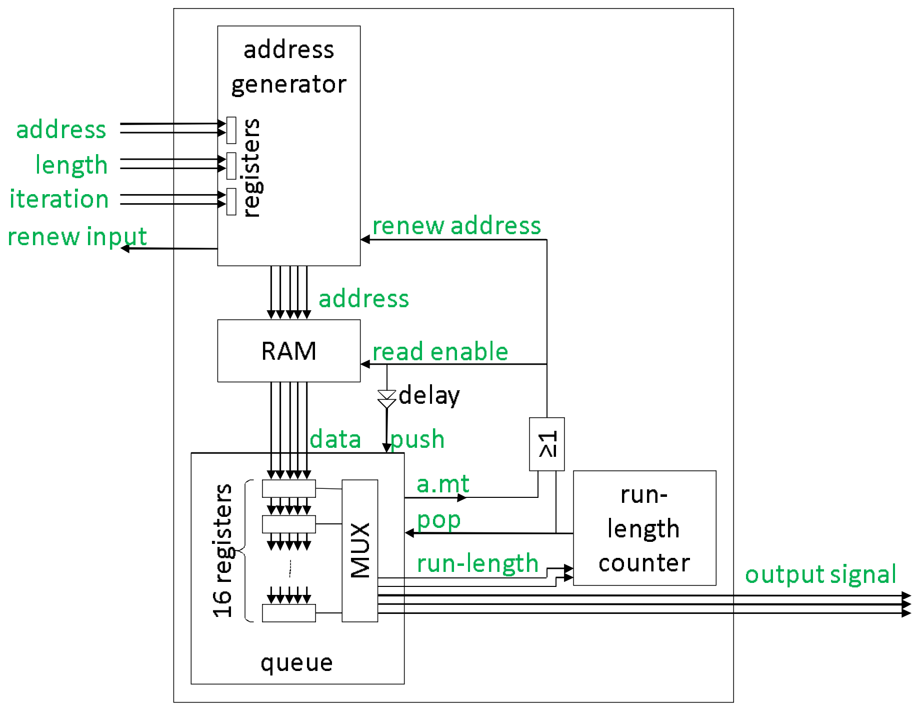


Figure A.4: The pattern generator

The address generation continues until the queue's "a.mt" (almost empty) flag goes reset. The almost empty flag is set when the number of contents in the queue is less than 8.

The value read out from the RAM is stored in the queue. Among the top value of the queue, 36 bits represent the waveform's bit pattern, while the rest 18 bits expresses the run-length. The run-length is passed to the run-length counter. The counter counts down the run-length, and when the count reaches 0, it generates a pop signal to the queue.

The queue, when a pop signal comes, discards the oldest value and change the output value to the next. The pop signal is given to the pattern memory (RAM) and the address generator, too.

After the address generator has all iterated the read-addresses, it immediately renews the input registers with the values on the input wires, which have been already renewed by the external module in response to the "renew input" signal at the beginning. At the same time it sets "renew input" again, and restarts yielding read-addresses. Thus the output signal is generated continuously.

The output stalls, however, only when the external module cannot respond to the "renew input" signal in time *and* the queue unexpectedly goes empty. Given the waveform fragment contains enough bit-patterns with long run-length, the queue will contain at least 8 values left when the address generator yields "renew input." The external module will then have a window time that long.

A.3.2 Pattern sequencer

The pattern sequencer has the program memory in which opcodes (Table A.1) are stored. The pattern sequencer reads the program memory and controls the pattern sequencer according to the program. The inner diagram is shown in Figure A.5.

In the pattern sequencer, there is also a RAM reader similar as the pattern generator. It is different from that of the pattern generator, however, in that it accepts jump addresses from the "loop" block. When a jump address is input, the RAM reader *resets* the queue and the shift registers for delay. The whole pattern sequencer stalls until the value at the jump address in the program memory comes out.

The operation code read out is provided to the three downstream blocks, "loop", "nop", and "CCD operation." Each of the three looks at the opcode, and executes the opcode if the opcode is targeted to it. It then sets its state as busy when the instruction takes two or more clocks.

The three states from the three blocks are multiplexed to be input to the "state" block. The "state" block yields veto, according to the input state, to prohibit execution of the next opcode.

The "loop" block has three tuples of a return address pointer and a loop counter. If `lstart` opcode comes, the "loop" block stores one plus the current instruction address to the return address pointer, and the loop count specified by the opcode to the specified loop counter. Every time `lcont` opcode comes,

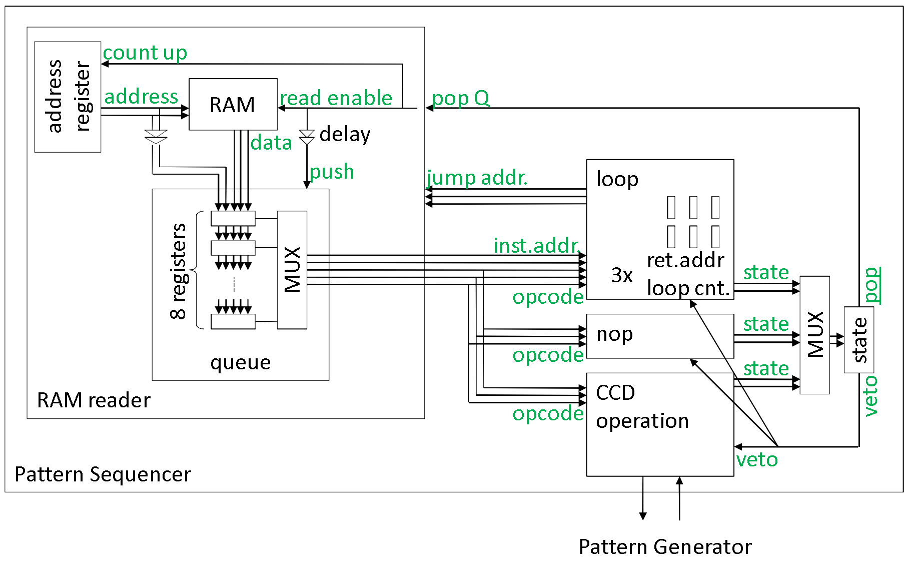


Figure A.5: The pattern sequencer

the "loop" block decrements the loop counter and outputs the return address as the jump address.

The "CCD operation" block communicates with the pattern generator located downstream. Its state is "busy" until the pattern generator accepts inputs and the renew input signal is set by the pattern generator.

A.4 Compiler

In the configuration of the clock generator, users are requested to prepare waveform data and program data. We made a compiler that translates source files into a binary file containing waveforms and programs. The syntax of the source files is designed to be compatible with Modularized Extensible System for Image Acquisition (MESSIA) series [11]. They are employed in the Suprime-Cam's readout system, and many other instruments in the Japanese astronomical community.

A.4.1 Source files

We show an example of the source codes in Figure A.6. The source codes are in the ASCII character coding.

```
# sample clock pattern definition

begin proc1                # procedure definition: "proc1"
    ccd_operation 0 1 100  # iterate ope "1" 100 times.
    ccd_operation 0 2 10   # iterate ope "2" 10 times.
end

begin proc2
    loop3_start 42_0000_0000
        ccd_operation 0 1 5
        ccd_operation 0 2 3
    loop3_continue
end

set_default_bit     1       # omitted bits are set 1
set_tick_prec       10      # tick precision is 10ns

set_clock_tick  3           # clock tick is 3 * 10ns
operation_type  1           # signal definition: ope "1"
start       35  32  0   5   4   10
t   4       |   |    ]   ]  |   |
t   1       |    ]  |    ]   ]  |
t   1       |    ]   ]   ]   ]  |
t   1       |    ]  |    ]  |   |
end

set_clock_tick  1
operation_type  2
start       35  32  0   5   4   10
t   2        ]  |    ]   ]   ]  |
end

#eof
```

Figure A.6: An example of clock pattern definition

Tokenization

The tokenization of the source codes goes as the following.

1. Spaces and line breaks
   Spaces and line breaks have different roles. Spaces are used to separate tokens and have no syntactic meanings, which line breaks have.

   - A sequence of carriage returns (0x0d) and line feeds (0x0a) are treated as a line break.
   - Any characters between a '`#`' and a line break are treated as a line break.
   - A sequence of whitespaces including at least one line break are treated as a line break.
   - A sequence of whitespaces without line breaks are treated as a space.

2. identifiers and keywords
   Identifiers and keywords are quarried with the same regular expression:
   `[A-Za-z_][A-Za-z0-9_]*`
   They are case sensitive.

3. Numbers
   Numbers are quarried with the regular expression:
   `[0-9][A-Za-z_0-9]*`
   Underscores can be arbitrarily inserted, which are just ignored in interpretation. Numbers are case insensitive. If the quarried token does not fit any of the following three, the compilation will fail.

   - Binary numbers: `[01][01_]*[Bb]`
     `0101_1100_b`, for example.
   - Decimal numbers: `[0-9][0-9_]*`
     `123` and `043` for example. The latter one is not an octal number.
   - Hexadecimal numbers: `[0-9][A-Fa-f0-9_]*[Hh]`
     '`ff`' (255) is expressed as '`0ffH`'. The leading zero must exist because '`ffH`' would be treated as an identifier.

   In addition, numbers must be expressed within 32-bit unsigned integer. Otherwise, the compilation will fail.

4. '`]`' and '`|`'
   '`]`' is interpreted as 1 and '`|`' is as 0 in waveform definitions.

Procedure definitions

Procedures are compiled to be program data. Their definitions have the following syntax:

begin *identifier linebreak*
commands
end *linebreak*

This sentence defines a procedure with the name *identifier*. *identifier* that begins with double underscores may be reserved for the compiler and should not be used.

commands is a sequence of the following commands:

- **ccd_operation** 0 *id count linebreak*
  id := *number*
  count := *number*
  Iterate *count* times a waveform fraction whose ID is *id*. The leading '0' is left for backward compatibility and has no meanings.
- **loop1_begin** *count linebreak*
- **loop2_begin** *count linebreak*
- **loop3_begin** *count linebreak*
  Loop *count* times. There are three counters, and up to triple loops can be expressed.
- **loop1_continue** *linebreak*
- **loop2_continue** *linebreak*
- **loop3_continue** *linebreak*
  Decrement the loop counter, and jump to the corresponding **loop*_begin** if it is still non-zero.
- **nop** $count_{\text{opt}}$ *linebreak*
  Stall for *count* clocks (or, *count* × 10 ns.) If *count* is omitted, it is assumed to be 1.

Waveform definitions

Before the waveform definitions, users should specify the length of a "tick," the unit of time used in waveform definitions. We prepared two keywords to specify the tick.

- **set_tick_prec** *number linebreak*
  Set the step size of the tick to be *number* nsec. This keyword is newly created for the clock generator. The default value is 80 because the corresponding value of MESSIA-V is fixed to 80. The minimal value of *number* is 10, and *number* must be a multiple of 10 because of the clock generator's hardware restriction.
- **set_clock_tick** *number linebreak*
  Set the tick to be *number* × the step size which is defined by **set_tick_prec**.

It is recommended that **set_tick_prec** is put at the beginning of a source file, and that **set_clock_tick** is put before every waveform definition. There is another option:

- **set_default_bit** *number linebreak*
  Set signals omitted in waveform definitions to be *number*/ *number* must be 0 or 1. The default value is 1.

It is recommended that **set_default_bit** is put at the beginning of a source file.

Then, a waveform fragment is expressed in the following syntax:

> **operation_type** *id linebreak*
> **start** *signals linebreak*
> *bitpatterns*
> **end** *linebreak*

The waveform is registered with ID *id*, and summoned by **ccd_operation** in procedures. *id* larger than `0_ffff_0000_H` may be reserved for the compiler and should not be used.

signals are a sequence of *number* representing bit IDs of signals in the range of 0–35. They can be arranged in any order, and unnecessary bits can be omitted. The omitted bits are set to be the value defined by **set_default_bit**.

bitpatterns are a sequence of *bitpattern*:

bitpattern := **t** *number levels linebreak*

where *levels* is a sequence of '`]`' and '`|`' representing signal level 1 and 0 respectively. If you print out the waveform definition and turn the sheet 90° counterclockwise, you see a parallel signal waveform.

Each bit pattern holds for *number* ticks, which was defined by **set_clock_tick**. The hold time becomes *number* $\times$ **clock_tick** $\times$ **tick_prec** nanoseconds. The hold time is internally represented by an unsigned 32-bit integer in tens of nanoseconds. Therefore the hold time divided by 10 ns must be $< 2^{32}$. If not, a compilation error occurs.

In addition, the hold time (divided by 10 ns) field, or the run-length field in the hardware has 18 bits only. The compiler automatically separates a longer bitpattern into many continuous ones with their run-length $< 2^{18}$.

Dividing source files

Source files can be divided on condition that neither procedure definitions nor waveform definitions are divided on their way. Procedure names and waveform IDs are global, while **set_default_bit**, **set_tick_prec**, and **set_clock_tick** are local.

A.4.2 Binary file format

An output binary file has a structure shown in Figure A.7. All fields are in network byte order. Applications that use the binary files should not assume that "Programs", "Clock patterns" and "Symbol tables" are stored in this order.

CPD_BINARY_HEADER
Programs
Clock patterns
Symbol table

Figure A.7: Binary file structure

CPD_BINARY_HEADER

CPD_BINARY_HEADER has the following fields.

offset	size	description
0	6	"CPD" 0D 0A 00
6	1	01 (Major version)
7	1	01 (Minor version)
8	4	File offset to the program data
12	4	Size of the program data
16	4	CRC32 (IEEE 802.3) of the program data
20	4	File offset to the clock pattern data
24	4	Size of the clock pattern data
28	4	CRC32 (IEEE 802.3) of the clock pattern data
32	4	File offset to the symbol table
36	4	Size of the symbol table
40	4	CRC32 (IEEE 802.3) of the symbol table

Programs

Programs are data to be stored in the program memory of the clock generator. This is a simple array of CPD_BINARY_PROGRAM:

offset	size	description
0	1	verb
1	4×1	arg1[4]
5	4×1	arg2[4]

where "verb" corresponds to that in Table A.1; "arg1" [0],[1],[2], and [3] corresponds to "argument 1" [31:24], [23:16], [15:8], and [7:0]; and "arg2" to "argument 2" in the same way.

Clock patterns

Clock patterns (waveforms) are data to be stored in the pattern memory of the clock generator. This is a simple array of CPD_BINARY_CLOCK:

offset	size	description
0	3×1	runlength[3]
3	5×1	pattern[5]

where “runlength” [0], [1] and [2] corresponds to “run-length” [17:16], [15:8], [7:0] in Figure A.2; and “pattern” [0], [1],..., [4] corresponds to “bit pattern” [35:32], [31:24],..., [7:0].

Symbol table

The symbol table is used by applications controlling the clock generator to look up procedure names. This is a simple array of CPD_BINARY_SYMBOL:

offset	size	description
0	62×1	name[62]
62	2	address

which indicates that the “name” procedure begins from the address in the program memory. The field “name” must be null-terminated, and thus its length ≤ 61.

A.4.3 Special procedure

In a binary output file, there is a special procedure generated automatically by the compiler. The symbol table contains a `__SET_BITS__` procedure which is associated with address 0 in the program data. The procedure generates the waveform data of length 1 at address 0 in the pattern memory only once, and soon exits. Note that addresses in the pattern memory have no relation to waveform IDs in source files and that it is *not* the waveform of ID 0 that is generated.

User-defined waveforms are written from address 1 in the pattern memory (the compiler generates binary a file whose the clock pattern region is already arranged so.) Rewriting address 0 of the pattern memory therefore affects no user-defined waveforms. Applications that control the clock generator can thus arbitrarily fix the bits of the signal output, rewriting address 0 of the pattern memory and invoking procedure at address 0 of the program memory.

Acknowledgements

I would like to thank my supervisor, H. Aihara. I owe a debt of gratitude to N. Katayama, R. Itoh, H. Furusawa and M. Takada. I have been given insightful advice from H. Kakuno, T. Abe, T. Uchida, and M. Iwasaki, to whom I am also grateful for her help in the emendation of this thesis. Additionaly, C. Ng kindly corrected my English expressions. I thank Y. Nakahama, H. Nakayama, H. Miyatake, H. Fujimori, S. Sugihara and J. Wang.

Bibliography

[1] http://www.i4s.co.jp/rcm/rcmabs.html.

[2] http://www.astromatic.net/software/scamp.

[3] http://www.astromatic.net/software/swarp.

[4] http://root.cern.ch/.

[5] http://www.mpi-forum.org/.

[6] http://www.mcs.anl.gov/research/projects/mpich2/index.php.

[7] http://mathema.tician.de/software/boostmpi.

[8] http://www.astromatic.net/software/sextractor.

[9] Andreas Albrecht et al. Report of the dark energy task force. http://arxiv.org/abs/astro-ph/0609591v1, 2006.

[10] G. Hinshaw et al. Five-year Wilkinson Microwave Anisotropy Probe (WMAP) observations: data processing, sky maps, and basic results. *Astrophysical Journal Supplement*, 180:225, 2009.

[11] H.Nakaya et al. New focal plane array controller for the instruments of the Subaru Telescope. *Publications of the Astronomical Society of the Pacific*, 118:478–488, 2006.

[12] H. Miyatake. Research and development of readout system for new wide-field ccd camera of subaru telescope, 2009.

[13] H. Miyatake et al. Prototype readout module for Hyper Suprime-Cam. In *2008 IEEE Nuclear Science Symposium Conference Record*, pages 737–741, 2008.

[14] S. Perlmutter et al. Measurements of Ω and Λ from 42 high-redshift supernovae. *Astrophysical Journal*, 517:565, 1999.

[15] A. Refregier. Weak gravitational lensing by large-scale structure. *Annual Review of Astron. Astrophys.*, 41:645, 2003.

[16] A. G. Riess et al. Observational evidence from supernovae for an accelerating universe and a cosmological constant. *Astronomical Journal*, 116:1009, 1998.