CRBLASTER: a fast parallel-processing program for cosmic ray rejection

Kenneth J. Mighell

aNational Optical Astronomy Observatory, 950 N. Cherry Ave., Tucson, AZ 85719, U.S.A.

ABSTRACT

Many astronomical image-analysis programs are based on algorithms that can be described as being embarrassingly parallel, where the analysis of one subimage generally does not affect the analysis of another subimage. Yet few parallel-processing astrophysical image-analysis programs exist that can easily take full advantage of todays fast multi-core servers costing a few thousands of dollars. A major reason for the shortage of state-of-the-art parallel-processing astrophysical image-analysis codes is that the writing of parallel codes has been perceived to be difficult. I describe a new fast parallel-processing image-analysis program called CRBLASTER which does cosmic ray rejection using van Dokkum’s L.A.Cosmic algorithm. CRBLASTER is written in C using the industry standard Message Passing Interface (MPI) library. Processing a single 800 x 800 HST WFPC2 image takes 1.87 seconds using 4 processes on an Apple Xserve with two dual-core 3.0-GHz Intel Xeons; the efficiency of the program running with the 4 processes is 82%. The code can be used as a software framework for easy development of parallel-processing image-analysis programs using embarrassing parallel algorithms; the biggest required modification is the replacement of the core image processing function with an alternative image-analysis function based on a single-processor algorithm. I describe the design, implementation and performance of the program.

Keywords: cosmic ray rejection, parallel processing, Message Passing Interface (MPI), CRBLASTER

1. INTRODUCTION

Astrophysicists have recently gained easy access to relatively inexpensive multi-processor computational hardware. An Apple Mac Pro computer with two 2.8-GHz quad-core Intel Xeon “Harpertown” processors starts at $2,799 and a 13” MacBook laptop computer with a 2.1-GHz Intel Core 2 Duo can be purchased for $1,099. Although ready access to multi-core platforms is now available, few parallel-processing astrophysical image-analysis programs exist that can easily take full advantage of these multi-processor computers.

A major reason for the shortage of state-of-the-art parallel-processing astrophysical image-analysis codes is that the writing of parallel codes has been perceived to be difficult. This perception, in general, is accurate. When writing parallel-processing software, the programmer frequently encounters problems associated with choreography of multiple processes which typically have complex interaction rules and requirements. Some of these interactions can cause problems never encountered with serial-processing software. For example, deadlock is a relatively simple condition that occurs when two processes hang because they both are waiting for the other to complete before proceeding. Livelock, on the other hand, is a much more complicated condition that occurs when two or more processes continually change their state in response to changes in the other processes and as a result none of the processes completes. Livelock frequently occurs when processes are too polite – like when two people meet in a hallway and each attempts to avoid the other by attempting to step around the other but they only move back and forth because they are always blocking each other’s progress. Professional-grade parallel-processing debugging software tools are complex and sophisticated because they must detect these and other dynamic programming errors in real time; not surprisingly, these tools may cost many thousands of dollars per license.
Programming problems can be classified as being *embarrassingly parallel* if they have computational workloads that can obviously be divided into a number of (nearly) independent parts that are executed on separate processes; each compute process does its work independently with no (or little) communication between other compute processes.  

Many low-level image processing operations involve only local data with very limited (if any) communication between areas of interest. Simple image-processing tasks such as shifting, scaling, and rotation are ideal candidates for embarrassingly-parallel computations. Similarly, for many astronomical image-analysis programs the analysis of one subimage does not affect the analysis of another subimage and these programs are thus excellent candidates for embarrassingly-parallel computation.

Embarrassingly-parallel image analysis of a single image requires that the image be partitioned for processing on the compute processes. Figure 1 shows a simple way that one may partition an image into 3 subimages for processing by embarrassingly-parallel algorithm on 3 compute processes. Partitioning the image over rows instead of columns would be logically equivalent.

During the conversion of a traditional single-process image-analysis program to an embarrassingly-parallel program, one must consider carefully edge effects where additional data beyond the edges of a particular image partition may be required in order to do a proper computation of the algorithm. In order to qualify as an embarrassingly-parallel computation, one tries to avoid interprocess communication between compute processes whenever possible; this make the parallel-processing program much easier to write and possibly faster to execute. Consider, for example, a 7×7 median filter. An embarrassingly-parallel implementation of the 7×7 median filter (using the partition scheme shown in Fig. 1) would broadcast subimages that include an overlap region of 3 pixels so that the computation of the 4th column from a common partition edge would be correctly computed; the data in the overlap region is used but the algorithm is only applied to the non-overlapping columns. For a 600×600 input image, the subimages would thus have sizes of 203×600, 206×600, and 203×600 pixels. But what is to be done with the first and last 3 columns of the input image and the first and last 3 rows of the subimages? The 7×7 median filter is not defined in those regions and the coding of the algorithm in those areas of the input image are thus implementation dependent.

---

**Figure 1.** A simple way to partition an image into 3 subimages for an embarrassingly parallel algorithm.
While the writing of parallel-processing codes is, in general, a challenging task frequently requiring complicated choreography of interprocessor communications, the writing of parallel-processing image-analysis codes based on embarrassingly-parallel algorithms, specifically, can be a much easier task to accomplish due to their limited need for communication between compute processes. In this article, I describe the design, implementation, and performance of an embarrassingly-parallel image-analysis program which does cosmic ray rejection of CCD images. The code has been designed to be used as a software framework which enables the easy development of other parallel-processing image-analysis programs based on embarrassingly-parallel algorithms.

2. CRBLASTER

I have developed a new fast parallel-processing image-analysis program, called CRBLASTER*, which does cosmic ray rejection using van Dokkum’s L.A.Cosmic² algorithm. CRBLASTER is written in C using the industry standard Message Passing Interface (MPI)†. Processing a single 800×800 Hubble Space Telescope (HST) Wide Field Planetary Camera 2 (WFPC2) image‡ takes 1.87 seconds using 4 processes on an Apple Xserve with two dual-core 3.0-GHz Intel Xeons; the efficiency of the program running with the 4 processors is 82%.

Figure 2 shows the flowchart diagram of CRBLASTER. The director process reads the input FITS image§, splits it into N subimages (with overlap regions), and then sends them to the actor processes. Each actor process (including the director process) does cosmic ray rejection using the L.A.Cosmic algorithm on their own subimage, and then sends the resulting cosmic-ray cleaned subimage to the director process. The director process then collects all of the cosmic-ray cleaned subimages and combines them together to form the output cosmic-ray cleaned image which has the same size of the input image.

*CRBLASTER is currently available at http://www.noao.edu/staff/mighell/crblaster
†See Appendix A for instructions on how to retrieve and build the latest version of the Open MPI implementation of the MPI library.
‡The WF3 dataset of the HST WFPC2 observation U3060302M.C0H of the galaxy cluster MS 1137+67.
§FITS images are read and written using version 3.06 of the FITS file format library which is included with the CRBLASTER software package. CFITSIO is currently available at http://heasarc.nasa.gov/docs/software/fitsio/fitso.html
The director process vertically partitions the input image into \(N\) subimages where \(N\) is the number of processes requested by the user. These subimages contain approximately \(\frac{1}{N^{th}}\) of the image plus an overlap region that is \(\text{BORDER}\) pixels high above/below all joint partition edges. For the L.A.Cosmic algorithm, the optimal value of \(\text{BORDER}\) has been determined to be 6 pixels; using less than 6 pixels leaves many cleaning artifacts, while using more than six pixels does not improve the quality of the final output image.

\texttt{crblaster} is a faithful implementation in C of the the IRAF\textsuperscript{6,7} script \texttt{lacos_im.cl}\textsuperscript{6} by Pieter van Dokkum. The L.A.Cosmic algorithm, as described in van Dokkum’s article\textsuperscript{6} or implemented in the IRAF script \texttt{lacos_im.cl}, does not have its behavior explicitly defined along the outer \(3\) pixels on each edge of the input image; the cosmic ray rejection of \texttt{crblaster} is very nearly a perfect implementation of \texttt{lacos_im.cl} (within the limits of the IEEE 754 floating point standard) except for the occasional pixel in the overlap regions which might be slightly off from the value produced by the IRAF script.

The current C-binding of the MPI standard library contains over one hundred functions\textsuperscript{3}, but embarrassingly-parallel programs can be written using only a minimal subset\textsuperscript{4,5} of the following six MPI functions:

- \texttt{MPI_Init()}: Initialize MPI communicator
- \texttt{MPI_Comm_size()}: Determine total number of processes in the communicator
- \texttt{MPI_Comm_rank()}: Determine the process number (rank) in the communicator
- \texttt{MPI_Send()}: Send data to another process
- \texttt{MPI_Recv()}: Receive data from another process
- \texttt{MPI_Finalize()}: Shut down communicator

An MPI version of the “Hello World” program (e.g., Listing 2 of Appendix A) calls only 4 of these basic MPI functions because such a program does not need to send/receive data to/from another process — unlike most embarrassingly-parallel programs.

The director/actor process(es) send/receive the subimages by using just two matching pairs of \texttt{MPI_Send()}/\texttt{MPI_Recv()} calls. The first pair sends/receives the subimage image structure as an array of \((\text{int})(\text{sizeof(struct imageS)}))\) bytes \((\text{MPI Datatype MPI_CHAR})\). This programming hack greatly simplifies the program but it does come at a cost: \texttt{crblaster} is assumed to be running within a homogeneous computing environment, which is a valid assumption for multi-core servers or Beowulf clusters composed of identical CPUs. The second pair of \texttt{MPI_Send()}/\texttt{MPI_Recv()} calls sends/receives the actual image data as an array of doubles \((\text{MPI Datatype MPI_DOUBLE})\). This generally works well when the subimage data is being sent to another process. However, if the subimage data is being sent from the director to itself (in its role as an actor), then some implementations of MPI might hang due to some assumptions of the maximum size of message any sane user would wish to self transmit to/from a given process. The subimage arrays may be many megabytes in size so it is prudent to replace the \texttt{MPI_Send()}/\texttt{MPI_Recv()} call with a simple memory copy (\texttt{memcpy}) whenever the director sends the subimage data to itself.

Running \texttt{crblaster} on just one process gives a speed improvement of a factor of about 5.8 over the IRAF implementation of the L.A.Cosmic algorithm. Figure 3 shows the measured performance of \texttt{crblaster} on a cluster with 20 processors. Note the nearly ideal improvement in the processing time of the core analysis function. The non-parallelizable portion of \texttt{crblaster} (mainly the reading/writing of the input/output images) could be greatly minimized by reading/writing the image data on/off a ramdisk instead of a physical hard disk drive — reading/writing from/to memory is much faster than to spinning magnetic disks.

The \texttt{crblaster} code can be used as a software framework for easy development of parallel-processing image-analysis programs using embarrassingly parallel algorithms. The biggest required modification is the replacement of the core image-processing function with an alternative image-analysis function based on a single-processor algorithm. If the new algorithm needs larger or smaller overlap region of the subimages, then the numerical value of the \texttt{BORDER} macro \([\#\text{define BORDER} (6)\) should be modified to the appropriate value for the new algorithm. And of course, the command line options will need to be modified to provide the new algorithm information about any required parameters. Beyond these simple modifications, nothing within the main software framework needs to be touched.

\textsuperscript{4}Currently available at http://www.astro.yale.edu/dokkum/lacosmic/lacos_im.cl
3. CONCLUSION

Many astronomical image-analysis programs are based on algorithms that can be described as being embarrassingly parallel, where the analysis of one subimage generally does not affect the analysis of another subimage. Yet few parallel-processing astrophysical image-analysis programs exist that can easily take full advantage of today’s fast multi-core servers costing a few thousands of dollars. A major reason for the shortage of state-of-the-art parallel-processing astrophysical image-analysis codes is that the writing of parallel codes has been perceived to be difficult. I described a new fast parallel-processing image-analysis program called CRBLASTER which does cosmic ray rejection using van Dokkum’s L.A.Cosmic algorithm. CRBLASTER is written in C using the industry standard Message Passing Interface (MPI) library. The code can be used as a software framework for easy development of parallel-processing image-analysis programs using embarrassing parallel algorithms; the biggest required modification is the replacement of the core image-analysis function (in this case the C-version of the L.A.Cosmic algorithm) with an alternative image-analysis function based on a single-processor algorithm.

Figure 3. The measured performance of CRBLASTER on a cluster with 20 processors. Wall time is the actual time of execution as measured on a clock on the wall. The (blue) circles gives the measured execution times for the core analysis function of CRBLASTER running with 1–10 and 20 processors. The (green) straight line gives expected speedup based on the ideal model of a purely parallel algorithm. The (red) curve gives the expected speedup based on a more realistic model that accounts for the transmission time of the subimages which include redundant pixels in the overlap regions. The left image is part of the WF3 dataset of the HST WFPC2 observation U3060302M.C0H of the galaxy cluster MS 1137+67; the right image shows the result of cosmic ray rejection using CRBLASTER (compare with Fig. 6b of van Dokkum’s article).
APPENDIX A. OPEN MPI

The Message-Passing Interface (MPI) was designed to be an industrial-strength message-passing environment that is portable across a wide range of hardware environments. MPI is now the de facto industry standard for message-passing parallel-programming model, replacing virtually all other message passing implementations used for production work. Most, if not all of the popular parallel computing platforms offer at least one implementation of MPI. Two portable implementations of the MPI library, that are freely available online, are MPICH and OpenMPI.

Listing 1 shows a Makefile for retrieving and building the latest version of the Open MPI implementation of MPI; this process takes about 13 minutes on an Apple PowerBook with 1.67-GHz PowerPC G4 processor.

```
# FILE: Makefile
# PURPOSE: Get and build OpenMPI from http://www.open-mpi.org
# AUTHOR: Kenneth John Mighell
# DATE: 2008MAY26
# SOURCE: http://www.noao.edu/staff/mighell/ompi/Makefile
#
# CAUTION: Replace all ,,,, with a TAB character to create a proper Makefile
#
PKG = OpenMPI
WEBSITE = "http://www.open-mpi.org/software/ompi/v1.2/downloads/"
TARBALL = "openmpi.tar.gz"
MY_OMPI_PREFIX = ${PWD}
VERSION = OMPI-VERSION
OMPI_FLAGS = --disable-mpi-f77 --disable-mpi-f90 --prefix=${MY_OMPI_PREFIX}
# OpenMPI expects that the CC environment variable is set to gcc
CC = gcc
CSH_FILE = ompi.csh
all: get unpack build csh

get:
...,
,...,echo "*****" ${PKG}: $@ "BEGIN ************************************"
,...,echo "get the tarball..."
,...,curl [WEBSITE] | grep " .tar.gz" | head -1 | cut -f 6 -d " " \
,...,TARBALL_LATEST
,...,echo "curl " [WEBSITE]/cat TARBALL_LATEST" >> $TARBALL"
,...,CURL_COMMAND
,...,echo "cat CURL_COMMAND"
,...,echo < CURL_COMMAND
,...,echo "*****" ${PKG}: $@ "END **************************************"

unpack:
...,
,...,echo "*****" ${PKG}: $@ "BEGIN ************************************"
,...,echo "unpack the tarball..."
,...,cat [TARBALL] | tar -xvf - > /dev/null
,...,echo "...done!"
,...,echo "determine version..."
,...,ls -1 | grep "openmpi" | egrep -v latest | head -1 > ${VERSION}
,...,echo "version is" 'cat ${VERSION}'
,...,echo "*****" ${PKG}: $@ "END **************************************"

build:
...,
,...,echo "*****" ${PKG}: $@ "BEGIN ************************************"
,...,echo "cd to" 'cat ${VERSION}'
,...,cd 'cat ${VERSION}' ; \
,...,echo "configure ${PKG} with ${OMPI_FLAGS} ..." ; \
,...,./configure ${OMPI_FLAGS} 2>&1 | tee ../myconfig.log ; \
,...,echo "...done!" ; \
,...,echo "build ${PKG} ..." ; \
,...,make all install 2>&1 | tee ../mymake.log
,...,echo "...done!"
,...,echo "*****" ${PKG}: $@ "END **************************************"

csh:
...,
,...,echo "*****" ${PKG}: $@ "BEGIN ************************************"
,...,echo "#!/bin/csh" > ${CSH_FILE}
,...,echo "setenv MY_MPI OPENMPI" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PREFIX $MY_OMPI_PREFIX" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH_OLD $MY_PATH_OLD" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH $MY_MPI_PATH" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH_OLD $MY_PATH_OLD" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH $MY_MPI_PATH" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH $MY_MPI_PATH_OLD" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH $MY_MPI_PATH" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH $MY_MPI_PATH" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH $MY_MPI_PATH" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH $MY_MPI_PATH" >> ${CSH_FILE}
,...,echo "setenv MY_MPI_PATH $MY_MPI_PATH" >> ${CSH_FILE}
,...,echo "# END"
```

Listing 1. Makefile for retrieving and building latest version of OpenMPI. Assumes existence of the make, gcc, curl, zcat, and tar commands along with the csh shell. All instances four commas need to be replaced with a TAB character in order to create a proper Makefile.

**Open MPI is currently available at http://www.open-mpi.org/**
To build Open MPI, either type the Makefile in Listing 1 or download†† it and initiate the build process by typing

```
make
```
on the command line. The entire retrieve/build process may well take 10 to 20 minutes to complete with a fast Internet connection.

Once the build process has ended, one gains access to Open MPI’s C compiler tool (mpicc) and run tool (mpirun) by sourcing the CSH shell script ompi.csh from the CSH command line:

```
source ompi.csh
```

This will fail if the environment variable LD_LIBRARY_PATH does not exist; if that is the case then type the command `setenv LD_LIBRARY_PATH` and source the shell script file once again.

To build the MPI version of the “Hello World” program, either type the hello.c program in Listing 2 or download‡‡ it and compile it MPI’s C compiler tool by typing

```
mpicc -o hello hello.c
```
on the command line.

```
/*
* FILE: http://www.noao.edu/staff/mighell/ompi/hello.c
* PURPOSE: MPI example of the “Hello World” program
* AUTHOR: Kenneth John Mighell
* DATE: 2008MAY26
* COMPIL: mpicc -o hello hello.c
* RUN: mpirun -np 3 ./hello
* RESULT: Hello World! (rank 0 of 3)
* Hello World! (rank 1 of 3)
* Hello World! (rank 2 of 3)
*/
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main (int argc, char **argv)
{
  int rankI, sizeI;
  MPI_Init(&argc, &argv); /* Start up MPI */
  MPI_Comm_size(MPI_COMM_WORLD, &sizeI); /* Get number of processes */
  MPI_Comm_rank (MPI_COMM_WORLD, &rankI); /* Get process rank */
  printf ("Hello World! (rank %d of %d)\n", rankI, sizeI);
  MPI_Finalize(); /* Shut down MPI */
  exit(0);
}
```

Listing 2. MPI version of the “Hello World” program.

Running the hello executable with 3 processes is done by requesting 3 processes with MPI’s run tool:

```
mpirun -np 3 ./hello
```
The output of the program with 3 processes should look like this:

```
Hello World! (rank 0 of 3)
Hello World! (rank 1 of 3)
Hello World! (rank 2 of 3)
```

††Currently available at http://www.noao.edu/staff/mighell/ompi/Makefile
‡‡Currently available at http://www.noao.edu/staff/mighell/ompi/hello.c
ACKNOWLEDGMENTS

I wish to thank Dr. John Samson of Honeywell Inc., Aerospace Systems for use of Figs. 1 and 2. This work has been supported by a grant from the National Aeronautics and Space Administration (NASA), Interagency Order No. NNG06EC81I which was awarded by the Applied Information Systems Research (AISR) Program of NASA’s Science Mission Directorate.

REFERENCES


