An I/O strategy for CM1 on Blue Waters

Leigh Orf <leigh.orf@ssec.wisc.edu>
Cooperative Institute for Meteorological Satellite Systems (CIMSS)
University of Wisconsin - Madison

Last updated November, 2015

Abstract

In preparation for creating unprecedented high resolution simulations of tornadic supercell thunderstorms on the Blue Waters petascale supercomputer, estimated to create several petabytes of data, an I/O strategy for the CM1 model is developed utilizing the core driver (enabling buffering files to memory) and compression, features of the HDF5 data format. The primary strategy is one which maximizes throughput to disk, minimizing the amount of wallclock time dedicated towards I/O. This document presents a description of the I/O strategy and how it fits in with the objectives of the NSF PRAC grant "Understanding Tornadoes and Their Parent Supercells Through Ultra-High Resolution Simulation/Analysis."

1 Introduction

CM1 is a cloud model developed by George Bryan of NCAR, and will be used to simulate supercells on Blue Waters. CM1 utilizes MPI in order to take advantage of the massively parallel architecture of modern supercomputers. In its current "off the shelf" form, cm1r16 (CM1, release 16) contains three output format options: GrADS, netCDF, and HDF5. However, none of these strategies in their current form scale well to hundreds of thousand of cores. A new strategy has been developed which is designed to maximize I/O throughput for proposed simulations utilizing $\sim$100kcores, hence minimizing the amount of wallclock time dedicated towards I/O.

Modern supercomputing architectures typically consist of multicore shared-memory modules. It is assumed that on Blue Waters, CM1 will be running 16 MPI processes per module, one on each "floating point core". A simple I/O strategy, which is the current HDF5 output option on cm1r16, is to have each MPI rank write its own piece of the full model domain to a single file during each write cycle. For a 100kcore simulation, this strategy would result in hundreds of millions of small files, creating an unwieldy situation for post-processing and visualization. Moreover, it has been found that this type of I/O strategy is inefficient, primarily due to latency, metadata server overhead, and general overhead associated with concurrently writing such a large number of files to disk on a shared resource.

Off-the-shelf cm1r16 also contains the option of writing one netCDF file to disk by collecting 2D slices of data to a designated root core and assembling the file from this core. This approach, however, scales poorly and results in I/O taking up the majority of wallclock time for a simulation exceeding only thousands of cores.

The strategy described herein came about via trial and error on machines such as the Kraken XSEDE supercomputer, HECToR, and the Blue Waters supercomputer.

2 Objectives

The objectives of the approach developed for this project are to

1. reduce the number of files written to a reasonable number
2. minimize the number of times you are doing actual I/O to the Lustre file system
3. write big files
4. make it easy for users to read in data for analysis and visualization after it is written

The first objective is accomplished by (a) having one file per node be written rather than one file per core, reducing the number of files by a factor of 16 on Blue Waters (b) letting files grow in memory as new time levels are added, such that each file contains O(100) time levels, further reducing the number of files by a factor of O(100) over the traditional method of one-file-per-MPI-rank and one-time-level-per-file. The second objective is accomplished by compressing data (several hdf5 compression options are available) and by holding off on writing to Lustre until memory has been sufficiently exhausted. The third objective is accomplished naturally as the result of the second, as the files are allowed to grow in memory as new time groups containing 3D data are written to until memory is scarce. Experience has indicated that Lustre throughput is generally maximized when you are writing "large" files (performance on Kraken and Blue Waters has been very good when each individual HDF5 file is on the order of 1-10 GB in size).

The last objective is accomplished through a software driver which contains an API that provides access to any subset of the model's full domain at any time. This API accepts as its input a top-level directory location, variable name, and gridpoint indices relative to the full model domain, for any 1, 2, or 3D subdomain of any saved field for a given time. This makes it such that end users do not need to concern themselves with the underlying directory and file layout or the details of how data are stored in each individual HDF5 file. This approach has simplified the process of creating a plugin for the VisIt visualization software.

3 Description

Using a process similar to that of the WRF model, cm1 specifies a list of run-time variables in a file called `namelist.input`. Within this file, a variable called `tapfrq`, specifying the frequency (in seconds) of history file output is assigned and read during the model's initialization phase before integration begins. Every `tapfrq` seconds in model time a subroutine called `writeout` is called. If the hdf5 option is chosen (also set in `namelist.input`) the hdf5 code described is activated.

3.1 Rank reordering

CM1 uses a 2D decomposition where each rank (core) contains a small 2D (east-west/north-south) piece of the total horizontal extent of the full model domain, and the entire vertical extent of the full model domain. On Blue Waters, each shared-memory module contains 16 cores which are organized in a 4 by 4 spatial grid.

In order for this approach to work, rank ordering must be explicitly specified such that MPI ranks are properly mapped to the hardware (this can also be done in software). On

Blue Waters, this can be done by specifying MPICH_RANK_REORDER_METHOD=3 in the PBS script submitted. This directs the scheduler to get rank ordering information from a file called MPICH_RANK_ORDER. On Blue Waters, the perftools module is loaded and the grid_order command is used to create the MPICH_RANK_ORDER file, such as:

```
grid_order -R -c 4,4 -g 100,100 > MPICH_RANK_ORDER
```

for a run with 10,000 MPI ranks with 16 ranks per core. The rank reordering makes it such that each shared-memory module contains a 3D subdomain of the full model domain mapped over the 16 cores on the module.

3.2 One I/O core per module

On each shared-memory module, one core, here referred to as an I/O core (local rank 0 on each shared-memory module) handles I/O in addition to computation. A MPI communicator is created for each module such that collective operations can be done spanning each module. This keeps collective I/O traffic from leaving the module where it would be more bandwidth limited. Ideally, memory would be addressed directly from the I/O core, requiring no MPI internode communication overhead, but that is beyond the scope of this approach (see Damaris).

When the writeout routine is called, each I/O core initializes its hdf5 file if it is the first time the routine has been called or if a new file needs to be created. If it is a "normal" write cycle (the most common case), all the directories have already been created and file and group handles are open and no initialization is needed. Then, the I/O core does buffered writes of metadata, and for each variable to be written, collects all 3D array data from all cores on its shared-memory module using MPI_Gather, reassembles the data into an array, and writes the 3D data to the file in memory.

3.3 Buffering to memory

This approach is possible due to the fact that CM1 has a rather small memory footprint for the proposed simulations, utilizing only $\sim$5% of available memory on each shared-memory module. Hence, a large amount of memory is available for other uses, and this includes buffering I/O to memory.

Trial and error has guided what threshold value for available memory should be selected before flushing to disk. Real-time observations of memory utilization on a 16 core cluster during the flushing process itself have revealed that there is a non-negligible amount of memory overhead required while data is being flushed, and that choosing too small a value for the threshold leads to memory swapping, drastically decreasing performance. This phenomenon requires further research and/or consultation with HDF5 programmers and Cray support.

Currently the code decides whether there is enough memory available to continue buffered writes by utilizing the Linux /proc filesystem to query /proc/meminfo on each module. This file contains information on how much memory is in available, free, cached, buffered, etc., on each shared-memory module. The module with the smallest amount of memory available is determined by calling MPI_REDUCE, and this is the number that is compared to the threshold that is supplied by the user.

Another approach is to simply set the number of times to buffer to memory to a constant value. Values used on Blue Waters typically range from 20-50, which should provide

plenty of memory headroom for a typical simulation - although outstanding issues with unexplained memory exhaustion continue to pop up.

3.4 Pseudocode and resulting directory and file structure

Pseudocode highlighting the new hdf5 write routine in cm1 follows:

```
1   do while model_time .lt. total_integration_time
2     integrate_one_time_step
3     if mod(model_time,tapfrq) .eq. 0 !Time to do I/O
4       if i am an io core
5         if first visit to this routine:
6           if iamroot: make directories
7           initialize core driver
8           open hdf5 file !these are all zero length, initially
9           create hdf5 groups
10        else if this is a new write to disk cycle:
11          close groups
12          close file !This flushes the file to disk
13          if iamroot: make directories
14          initialize core driver
15          open hdf5 file !these are all zero length, initially
16          create hdf5 groups
17        else:
18          close current time group
19          create new time group
20        end if
21        write metadata and location data
22        MPI_Gather 3D data to io core from other cores on shared memory module
23        Reassemble 3D data on I/O core
24        write 3D data to time group
25      end if
26    end if
27  end do
```

This new hdf approach utilizes a strict organizational structure for directories and file locations, as well as the naming of files and directories. Directories and subdirectories are created as needed, using the FORTRAN `system` function to call the Linux shell command `mkdir`, from rank 0. Files are organized as follows:

`[basedir]/history/3D/[base].TTTTT/MMMMMM/[base].TTTTT_NNNNNN.cm1hdf5`

where:
[basedir] is `output_path`, set in `namelist.input`
[base] is `output_basename`, set in `namelist.input`
TTTTT is zero-padded model time of first time chunk
MMMMMM is a zero-padded directory. The number of files in this directory is set in `orfmod.F`.

NNNNNN is a zero-padded number refering to the "rank" of the shared-memory module that wrote the file.

An example directory listing from a 24576 core simulation on Kraken:

```
krakenpf2:~/scratch/cm1r16orf/den1hr% ls -R history
history/3D:
./              juice13.01380/  juice13.04080/  juice13.06360/
../             juice13.02420/  juice13.04840/  juice13.07060/
juice13.00000/  juice13.03320/  juice13.05640/

history/3D/juice13.00000:
./  ../  000000/  001000/  002000/

history/3D/juice13.00000/000000:
./                              juice13.00000_000499.cm1hdf5
../                             juice13.00000_000500.cm1hdf5
juice13.00000_000000.cm1hdf5    juice13.00000_000501.cm1hdf5
juice13.00000_000001.cm1hdf5    juice13.00000_000502.cm1hdf5
.
.
.
```

This directory and file naming convention is exploited in driver code (described below) that provides an API for providing easy access to model data spread across these files.

4 Structure of .cm1hdf5 files

HDF5 is an advanced self-describing scientific data format. HDF5 was chosen for this project instead of other self-describing data formats (such as netCDF) because of advanced features not found in other formats, such a hierarchical group structure, transparent lossy and lossless data compression, and the ability to buffer files to memory.

Each hdf5 file contains information about itself (such as where it lies within the full model domain) as well as basic information like grid spacing, the number of grid points in each dimension in the file, etc. This information is stored in the grid and mesh groups, i.e.:

```
% h5ls -r juice13.07060_002000.cm1hdf5  | egrep '(grid|mesh)'
/grid                    Group
/grid/myi                Dataset {1}
/grid/myj                Dataset {1}
/grid/ni                 Dataset {1}
/grid/nj                 Dataset {1}
/grid/nodex              Dataset {1}
/grid/nodey              Dataset {1}
/grid/nx                 Dataset {1}
/grid/ny                 Dataset {1}
/grid/nz                 Dataset {1}
/grid/x0                 Dataset {1}
/grid/x1                 Dataset {1}
```

```
/grid/y0                 Dataset {1}
/grid/y1                 Dataset {1}
/mesh                    Group
/mesh/dx                 Dataset {1}
/mesh/dy                 Dataset {1}
/mesh/xf                 Dataset {60}
/mesh/xh                 Dataset {60}
/mesh/xhfull             Dataset {1920}
/mesh/yf                 Dataset {30}
/mesh/yh                 Dataset {30}
/mesh/yhfull             Dataset {1920}
/mesh/zf                 Dataset {281}
/mesh/zh                 Dataset {280}
```

Each hdf5 file contains these data which provide information on the full model domain as well as the file's subdomain. Any single hdf5 file can therefore be interrogated for basic domain information, which can then be used to to assemble any subdomain of the model's full domain, spanning one or many individual files. It should be noted that the following grid variables are always constant among all files: `ni, nj, nodex, nodey, nx, ny, nz`. Grid variables which are unique for each hdf5 are `myi, myj, x0, x1, y0, y1`. The mesh group contains variables and 1D arrays that describe the model's mesh (units of meters), both of the file and of the full model domain, the latter of which is identical for all hdf5 files.

The 3D floating point files are stored as follows, which lists two adjacent time levels in one hdf5 file:

```
krakenpf2:002000% h5ls -r juice13.07060_002000.cm1hdf5  | grep 071[02]0
/07100                   Group
/07100/2d                Group
/07100/3d                Group
/07100/3d/dbz            Dataset {280, 30, 60}
/07100/3d/pipert         Dataset {280, 30, 60}
/07100/3d/prspert        Dataset {280, 30, 60}
/07100/3d/qc             Dataset {280, 30, 60}
/07100/3d/qg             Dataset {280, 30, 60}
/07100/3d/qi             Dataset {280, 30, 60}
/07100/3d/qr             Dataset {280, 30, 60}
/07100/3d/qs             Dataset {280, 30, 60}
/07100/3d/qv             Dataset {280, 30, 60}
/07100/3d/rhopert        Dataset {280, 30, 60}
/07100/3d/thpert         Dataset {280, 30, 60}
/07100/3d/uinterp        Dataset {280, 30, 60}
/07100/3d/vinterp        Dataset {280, 30, 60}
/07100/3d/winterp        Dataset {280, 30, 60}
/07100/3d/xvort          Dataset {280, 30, 60}
/07100/3d/yvort          Dataset {280, 30, 60}
/07100/3d/zvort          Dataset {280, 30, 60}
/07120                   Group
```

```
/07120/2d               Group
/07120/3d               Group
/07120/3d/dbz           Dataset {280, 30, 60}
/07120/3d/pipert        Dataset {280, 30, 60}
/07120/3d/prspert       Dataset {280, 30, 60}
/07120/3d/qc            Dataset {280, 30, 60}
.
.
.
```

The top level group is a zero-padded integer (cast to a character string) indicating the model time in seconds. Beneath this is the 3d group (2d data can be stored in these files as well), followed by the name of the variable. The number of time groups in any hdf5 file typically depends on how much data was buffered into memory before it was written to disk. The reading routines, described below, are flexible enough such that this number can vary between dump intervals; e.g., the first dump may contain 53 time levels, the second 47, etc. This variation has been observed when using a fixed memory threshold value for determining when to flush to disk (such as that describe above regarding the `/proc` filesystem).

5 Accessing model data for post-processing and visualization

The approach described above was chosen over other methods via trial and error with the primary goal to minimize the amount of wallclock time spent doing I/O. Since the output is spread over many files containing multiple time levels, some effort is required to reconstruct the model's full domain for post analysis and visualization. A driver has been written which simplifies this process.

Post-processing and visualization will require that the model's domain, or a subset of it, be read into arrays which are pieces of the model's full domain without concern for how they are spread across history files. These pieces are defined by grid indices relative to the model's full domain, in the space {1:nx,1:ny,1:nz}. The driver described below will return any 3D, 2D, or 1D piece of the model's full 3D domain, requiring only the grid indices of the desired space, so long as it is a subspace of {1:nx,1:ny,1:nz}.

The driver code is written in C and is serial (however, since reading in data is "embarrassingly parallel" the routine may be called from any MPI rank). Given the location of the history directory as input, the code will first traverse the directory structure and identify the layout. Then, it will interrogate the first hdf5 file it identifies and read in grid data (recall that some grid data in the hdf5 files is the same across all files). Next, a call is made to the routine that does all the "heavy lifting" and fills a buffer with the requested data. The listing below contains calls taken from C code which utilizes the driver code, in this case to read in model data and write a netCDF file:

```
1   get_sorted_time_dirs(topdir,timedir,times,ntimedirs);
2   get_sorted_node_dirs(topdir,timedir[0],nodedir,&dn,nnodedirs);
3   get_all_available_times(topdir,timedir,ntimedirs,nodedir,&ntottimes);
4   get_first_hdf_file_name(topdir,timedir[0],nodedir[0],firstfilename);
5   get_hdf_metadata(firstfilename,idum,idum,&nx,&ny,&nz,&nodex,&nodey);
6   read_hdf_mult_md(buffer,topdir,timedir,nodedir,ntimedirs,dn,
```

```
7       times,itime,varname,X0,Y0,X1,Y1,Z0,Z1,nx,ny,nz,nodex,nodey);
```

In lines 1-3 of the above listing, information is gathered about the directory structure and the file layout. These only need to be called once, as this information is static for any one simulation. Line 4 is a routine which returns the full path of any one available hdf5 file, with the intent to read in data (such as grid information) that is identical in all hdf5 files (line 5 gets such "metadata"). At this point, we have all the needed information to read in any subdomain at any available time. In line 6, requested data is read into a buffer. It is up to the calling routine to convert that buffer (a 1D floating point array) into a 2D or 3D array if need be, or address the values of the 1D array properly.

A description of the arguments of `read_hdf_mult_md` follows:

```
void read_hdf_mult_md(float *buffer, char *topdir,
char **timedir, char **nodedir, int ntimedirs,
int dn, int *times, int itime, char *varname,
int X0, int Y0, int X1, int Y1, int Z0, int Z1,
int nx, int ny, int nz, int nodex, int nodey);
```

`buffer` : The floating point buffer to contain the requested data. Caller must allocate data beforeheand.

`topdir` : The top level directory (should always point to whatever directory contains the directory called `3D`). This must be provided by the calling program.

`timedir` : a character array containing the names of the directories in the 3D directory. These are sorted. This has already been set by calling `get_sorted_time_dirs`.

`nodedir` : a character array containing the names of the directories in the `timedir` array. These are sorted. This has already been created by `get_sorted_node_dirs`.

`ntimedirs` : The number of time directories. Set in `get_all_available_times`.

`dn` : The integer interval between the numerical values of the zero-padded "node" directories. Set in `get_sorted_node_dirs`. In the `ls -R` listing above, this is 1000.

`times` : An integer array containing the times contained within the name of the time directories. In the `ls -R` listing above, this would be [0, 1380, 2420, 3320, 4080, 4840, 5640, 6360, 7060]. This was set in `get_sorted_time_dirs`.

`itime` : The requested time. This must be provided by the calling program, and be an actual time saved in the files.

`varname` : The name of the variable to be read.

`X0,Y0,Z0,X1,Y1,Z1` : The indices of requested piece of the full model domain. Must lie within {1:nx,1:ny,1,nz}. Does not need to be a 3D space (i.e., a horizontal slice would be retrieved if `Z0=Z1`).

`nx,ny,nz` : The number of grid points in x, y, and z for the full model domain (set in `namelist.input` and saved in all hdf5 files).

`nodex,nodey` : The node number (each node maps to one file) in X and Y for the 2D decomposition (not to be confused with nodex and nodey in namelist.input which refers to the MPI rank in X and Y).

The main point to realize with this approach is that the routine calling the driver code is "hidden" from the actual directory structure, the way data is spread across files, the file naming convention, and structure within the hdf5 files. This makes it simple to read in any piece of the full model domain, realizing that we don't always need to read in the full domain for analysis and plotting. Thus far, routines utilizing the driver code that have been written include routines that create single netCDF files and single Vis5D files, as well as scripts which create NCAR Vapor netCDF files. Example applications of this method for very large simulations include creating netCDF files for viewing with the ncview slice viewer (which can be run remotely on the supercomputing login node and displayed to a workstation in your office), or creating Vis5D files of targeted regions of the storm, such as just the storm's updraft, tornado, etc., realizing that Vis5D cannot display very large data sets.

This approach has proved very useful is in the construction of a VisIt plugin for rendering the full storm and works very well on all platforms tested thus far. It enables any aribtrary decomposition to be applied when visualizing and does not require VisIt domains to lie on file boundaries. Each rendering core only reads what is determined at run time, based upon a simple decomposition strategy that is based upon the number of processing cores available. This process has been extended to work seamlessly with subdomains of the full model domain in order to focus resources only on the areas of the storm of interest.

6 2D writes

All of the above discussion has concerned so-called `.cm1hdf5` contained in the `3D` directory. These files are primarily comprised of 3D arrays of selected variables. One of the objectives of this work was to create an up-to-date web page with contour plots at pre-specified heights above ground of the current state of the model while it is running on Blue Waters. Such plots contain 2D horizontal slices, consistent with typical meteorological maps. For quick analysis and monitoring a running simulations, it is very advantageous to have files containing only 2D fields of horizontal levels available in a file, rather than have to create 2D fields on the fly by reading in such data from the 3D files. Experience has shown that constructing 2D plots spanning the full model domain is very I/O intensive and hence time consuming, especially if it's done serially.

Therefore an option is provided in the new hdf5 routines of cm1 where so-called 2D files are created (these have the prefix `.2Dcm1hdf5` and are stored in the `2D` directory). By setting the Boolean variable `wr2d` to true, so-called 2D files are written during I/O cycles. These 2D files are written using parallel HDF5, and only one file is written per time dump. The code is written such that either 2D writes, 3D writes, or both can be chosen during any write cycle, based upon the Boolean variables `wr2d` and `wr3d` being set to `.true.` or `.false.`. Experience has shown that writing a single file doing parallel I/O (all-to-one) does not require an inordinate amount of wallclock time so long as small "patches" of data are supplied from each MPI rank. In the case of 2D horizontal slices for the proposed simulations, each MPI rank will be supplying a 2D array on the order of 15x15 grid points. pHDF5 handles this automatically in such a way as to reduce the write to only several writers rather than having all writers access the file, which would cause performance degradation.

Since each of the variables in the 2D files are at a single Z level, the files end up being a reasonable size (several hundred times smaller than an equivalent directory filled with `.cm1hdf5` files). These files can therefore be copied over to whatever machine is hosting

the web page, and the plotting routines should be able to run on the same machine as it will only require a machine of modest means to render the 2D plots serially, only needing to read from a single file.

6.1 Why not use pHDF5 for 3D writes?

There are several reasons why serial hdf5 was chosen for the 3D files rather than pHDF5, as is chosen for the 2D files. They include:

1. It is too slow to do it this way, especially if one or only a few files are written
2. You cannot buffer writes to memory, hence you must hit the Lustre file system for each write
3. You cannot use compression with pHDF5, hence you will have larger files and need to hit the Lustre file system more often
4. Such files would be very large

Incidentally, a nice side effect of spreading the 3D domain over many files is that it makes it possible to move a small subset of files for a given time off of Blue Waters for rendering, analysis, etc. For tornadic supercell thunderstorm simulations, the region of interest regarding research objectives is usually centered around the supercell updraft and downdrafts. This portion of the storm typically spans a small (say, around 5%) of the full horizontal extent of the model domain. Realizing that the simulations created on Blue Waters will likely be analyzed for years to come, it is therefore valuable to have the ability to do off-site analysis and visualization on more modest hardware. The reading routines described above do not require that the entire file set be present when reading in data, so long as the subset of the full model domain requested does not lie outside of the bounds of the files that are saved (this is why there is a routine called `get_first_hdf_file_name` rather than expecting any single file to exist).

7 Concluding remarks

An I/O strategy has been developed and has been shown to work for the CM1 model on the Blue Waters supercomputer. This strategy utilizes some of the advanced features of the HDF5 scientific data format in order to reduce I/O wallclock time to a reasonable percentage of total integration time. A driver has also been written which handles this data format natively, only requiring the calling program to specify the requested subdomain (or full domain) spanning the hdf5 files containing 3D data. The work already completed fulfills some of the major objectives of the Wilhelmson group's PRAC, and partly fulfills other objectives.

8 Appendix A: Example performance on Kraken

The following CM1 output contains timing data averaged for all processes on a 24576 core CM1 simulation on Kraken. In this simulation, I/O (which includes 3D writes, 2D writes, intermediate calculations for derived 3D variables, and flushing to the Lustre file system) comprised about 13% of wallclock time. Recent simulations on Blue Waters have indicated

similar time spent on I/O for some simulations (the percentage increases as I/O is more frequent, as expected). See cm1 documentation and source code for descriptions of these quantities.

```
 Total time:     24799.91

   sound    :     4920.58   19.84%
   mp_total :     4794.67   19.33%
   write    :     3242.24   13.07%
   stat     :     2857.74   11.52%
   advs     :     2734.44   11.03%
   microphy :     1163.13    4.69%
   tmix     :     1146.95    4.62%
   misc     :      637.85    2.57%
   pdef     :      576.07    2.32%
   turb     :      464.83    1.87%
   diffu    :      452.81    1.83%
   advw     :      411.51    1.66%
   buoyan   :      374.32    1.51%
   integ    :      286.07    1.15%
   advv     :      235.56    0.95%
   advu     :      209.25    0.84%
   prsrho   :      122.50    0.49%
   satadj   :       66.88    0.27%
   rdamp    :       53.55    0.22%
   divx     :       34.92    0.14%
   bc       :       13.93    0.06%
   sfcphys  :        0.12    0.00%
   swaths   :        0.00    0.00%
   radiatio :        0.00    0.00%
   poiss    :        0.00    0.00%
   pbl      :        0.00    0.00%
   parcels  :        0.00    0.00%
   fallout  :        0.00    0.00%
   cor      :        0.00    0.00%

   mps2     :     3141.94   12.67%
   mps1     :      657.05    2.65%
   mpu2     :      392.97    1.58%
   mptk1    :      384.73    1.55%
   mpw2     :       87.59    0.35%
   mpv2     :       35.52    0.14%
   mpq2     :       25.12    0.10%
   mpv1     :       22.39    0.09%
   mpw1     :       22.17    0.09%
   mpu1     :       20.97    0.08%
   mpq1     :        4.20    0.02%
   mptk2    :        0.00    0.00%
```

```
mpp2      :         0.00    0.00%
mpp1      :         0.00    0.00%
mpb       :         0.00    0.00%
```