Performance Comparison

between

ROOT, Objectivity/DB and LHC++ histOOgrams

René Brun and Fons Rademakers#*

Introduction

This note describes the first results of a performance benchmark we recently did between ROOT [1], Objectivity/DB [2] and the LHC++ histOOgram package [3]. In this benchmark the performance of each of these systems is tested in a typical physics data analysis (data-mining) environment. The tests were done on a Monte Carlo data sample, originally produced in HBOOK [4] Ntuple format, by the ATLAS collaboration. The main quantities measured were database traversal and histogramming performance and database size.

The Database

The original database used for this test was a set of 10 HBOOK Ntuple files each of about 18.4 MBytes. Each file contained a Column-Wise Ntuple (CWN) with 1000 Monte Carlo events. Each event had 91 attributes and 13 variable length arrays describing each particle of the event (on average 567 particles per event). Due to some compression performed by the HBOOK CWN system the total of 184 MB's was equivalent to about 270 MB of uncompressed data.

From HBOOK to Objectivity database

Based on the original Ntuple definition an Objectivity DDL (Data Definition Language) file was produced describing the database schema. The Ntuple event became an `Event` class with 76 attributes, a fixed length array of 3 `Calorimeter` objects (of 2 attributes) and one `HepVArray` containing the variable number of particles. The particles were described by a `Particle` class with 13 attributes. In addition one `Tag` class with 9 attributes was defined. The idea of this `Tag` class is that for each event a small `Tag` object is written to a separate database file and that during analysis the main selections are done using the `Tag` attributes. Since the `Tag` objects are small they can be read

*Rene.Brun@cern.ch, CERN.
#Fons.Rademakers@cern.ch.

efficiently from the database (and in many cases cached in memory) and therefore considerably speed up the analysis process. Using a preprocessor the DDL was converted to a set of header files containing the C++ classes.

Next the 10 HBOOK files were converted to a single large ASCII file which was subsequently read by a small C++ program that produced the Objectivity database. The resulting Objectivity database consisted of the following 5 files (size in bytes):

```
(shift45) #[150] ls -l  /shift/shd09/data10/vp/users/rdm/objy
-rw-r--r--   1 rdm        vp              228 Nov 20 12:58 ATLASPROD
-rw-r--r--   1 rdm        vp           460800 Nov 21 20:49 ATLASPROD.FDDB
-rw-r--r--   1 rdm        vp        274298880 Nov 19 17:59 Events.ATLASPROD.DB
-rw-r--r--   1 rdm        vp            71680 Nov 19 17:59 System.ATLASPROD.DB
-rw-r--r--   1 rdm        vp           755712 Nov 19 17:59 Tag.ATLASPROD.DB
```

The `Event` and `Particle` objects were stored in the Events.ATLASPROD.DB file while the `Tag` objects were stored in the file Tag.ATLASPROD.DB. The other 3 files are control files created by Objectivity and the HEP OODBMS layer. The used database page size was 2 KB. From the sizes we can see that Objectivity does not perform any compression. In fact the total database size is 50% larger than the equivalent HBOOK Ntuple files. Also, whenever one wants to transfer a database from one machine to another all 5 files need to be ftp-ed or copied and the database has to be reconfigured using the program *ooinstallfd*.

From HBOOK to ROOT database

The initial step of going from HBOOK to a ROOT database was done automatically via the program *h2root*. The *h2root* utility converts all objects in a HBOOK file to equivalent objects in a ROOT file. This initial step resulted in a single ROOT database of 88 MB. However, to emulate as closely as possible the ATLAS Objectivity class structure we introduced an equivalent set of `Event`, `Calorimeter` and `Particle` classes and converted the simple `TNtuple` object (which was created by *h2root*) to a more complex `TTree` object which contained the `Event`, `Calorimeter` and `Particle` objects for all events. In the ROOT case the `Event` class uses a `TClonesArray` object to store the variable number of particle objects. Furthermore, we did not create the small tag database since objects in ROOT `TTree` containers can be addressed directly per event and per attribute, thereby removing the need for the tags[1]. The resulting ROOT database consisted of the following single file (size in bytes):

```
(shift45) #[153] ls -l  /shift/shd09/data10/vp/users/rdm/root
-rw-r--r--   1 rdm        vp         84291475 Nov 22 12:58 event.root
```

From the size we can see that ROOT performs a reasonable amount of compression. In fact the total database size is 50% smaller than the equivalent HBOOK Ntuple files and **3 times smaller** than the equivalent Objectivity database. The `TTree` block size was 32 KB (optimal for the very large databases ROOT was designed for). A ROOT

[1] Of course we could also create a tag database, but this makes only sense when it is used to reference objects residing in large databases that migrate to mass storage devices. However, during the conversion exercise we also created a ROOT tag database containing the defined `Tag` objects. This tag database was exactly 29.9 KB (the high compression ratio is due to the fact that most `Tag` attributes are integers). This ROOT tag database was not further used during the tests.

database is machine independent and can be ftp-ed and copied between any platform without the need for a setup or configure program, or it can be accessed remotely via the `rootd` daemon.

Benchmarking Environment

All tests were performed on the CERN shift45 machine. Shift45 is a HP PA-RISC 8000 machine with 256 MB of RAM and 4 CPU's. The program files resided in the user directory under AFS and the databases were on an NFS mounted disk from shift9 (raw read performance of 2.5 MB/s over FDDI). At the time of the tests shift45 and shift9 were empty. Objectivity's lockserver, *ools*, was running on shift45.

The version of Objectivity was 4.0.2, the version of histOOgrams was the one from /afs/cern.ch/sw/lhcxx/specific/@sys/HISTOOGRAMS/1.0/, and the ROOT version was 1.03/07. All test programs were compiled with HP's CC compiler and the significant compiler options were "+a1 -O".

All reported timings are the real, user perceived, wall clock time as reported by `/bin/time`. Each measurement was repeated three times in a row.

Database Traversal Performance

To measure the database traversal performance of Objectivity and ROOT a small test program was used that calculated the total particle mass for events with a total number of particles below a certain limit. Therefore the main loop of the program looped over all 10000 events in the database and for the events with the attribute `Npart`<*MaxPart* there was a second loop over all particles summing the mass of each particle. This test effectively only accessed one attribute of each `Event` object, `Npart`, and one attribute, `mass`, of each `Particle` object. In the case of Objectivity the attribute `Npart` was in the small tag database to speed up analysis. Both test programs can be found in the Appendices A and B of this note.

Traversal Results

The results of the database traversal test are summarized in the table below.

MaxPart	Events	Objectivity	ROOT
	(in percent)	*(in seconds)*	*(in seconds)*
0	0.0	3.5	1.8
400	4.8	13.2	5.9
450	13.6	38.9	7.2
500	27.5	52.8	8.1
550	45.4	78.9	12.1
600	63.8	88.4	16.1
650	79.2	98.1	17.9
10000	100.0	122.0	18.2
Total time		495.8	87.3

Table 1: Database traversal performance

Each timing is the average of three consecutive runs. Typically the first run took up to 50% longer than the other two runs. Especially with a small *MaxPart* value, in which case the selected particle mass data could be cached locally by the system.

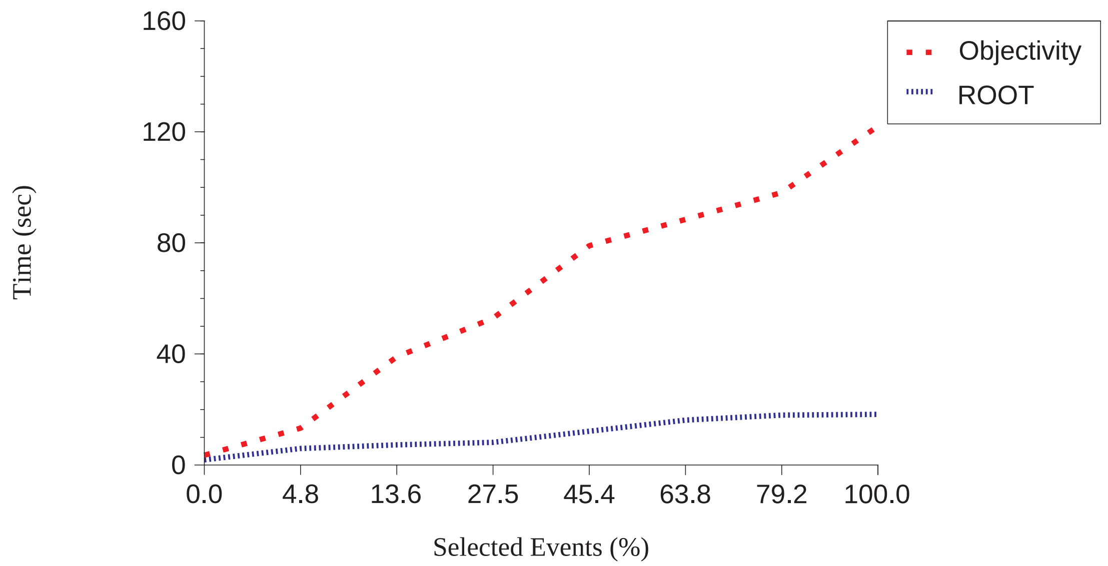

Figure 1: Database traversal performance

From the results it seems that Objectivity reads for every selected event all `Event` and all `Particle` attributes (although some clustering hints were specified when the database was created). ROOT's `TTree` container, on the other hand, allows direct access to each individual `Event` and `Particle` attribute (attribute access can be controlled at database read time). The advantage of this approach is clearly reflected in the superior performance numbers of the ROOT database system.

Database + Histogramming Performance

In this test we compared the histogramming performance of ROOT and the LHC++ histOOgram package. We used the programs from the previous test and added code for the creation, filling and storing of 3 histograms of 100 bins. The first histogram histogrammed `Npart`, the second the mass attribute of each particle and the third the total mass, i.e. the sum of the particle masses. At the end of the program the histograms were made persistent either in a ROOT or Objectivity database. In the 0% runs only `Npart` was histogrammed, 10000 entries, while in the 100% runs the total number of entries for the 3 histograms was 5691244 (10000, 5671244, 10000 entries, respectively). We tested the following combinations: Objectivity+histOOgram, Objectivity+ROOT histograms and pure ROOT (database+histograms). In all cases we verified that the results were identical.

Database + Histogramming Results

The table below shows the combined timing of database traversal and histogramming for each of the three above mentioned combinations. From the results in the table we see that the combination Objectivity+histOOgram performs very poorly, more than a factor 2.5 worse than Objectivity+ROOT and about a factor 14 compared to the pure ROOT solution. From these results we see that histOOgrams suffer a large overhead during filling, either caused by a too complex class structure, a bad filling algorithm, a large Objectivity overhead or a combination of the three.

MaxPart	Events *(in percent)*	Objy/histOO *(in seconds)*	Objy/ROOT *(in seconds)*	ROOT *(in seconds)*	Entries
0	0.0	4.5	3.9	1.9	10000
400	4.8	16.1	13.5	6.2	186650
450	13.6	61.6	39.3	7.6	564738
500	27.5	92.1	54.8	9.6	1224797
550	45.4	150.0	81.3	13.8	2168384
600	63.8	204.0	92.4	19.0	3225376
650	79.2	255.0	103.0	21.7	4186002
10000	100.0	328.0	129.0	23.4	5691244
Total time		1111.3	517.2	103.2	

Table 2: Database + histogramming performance

Graphically this behaviour is even more striking.

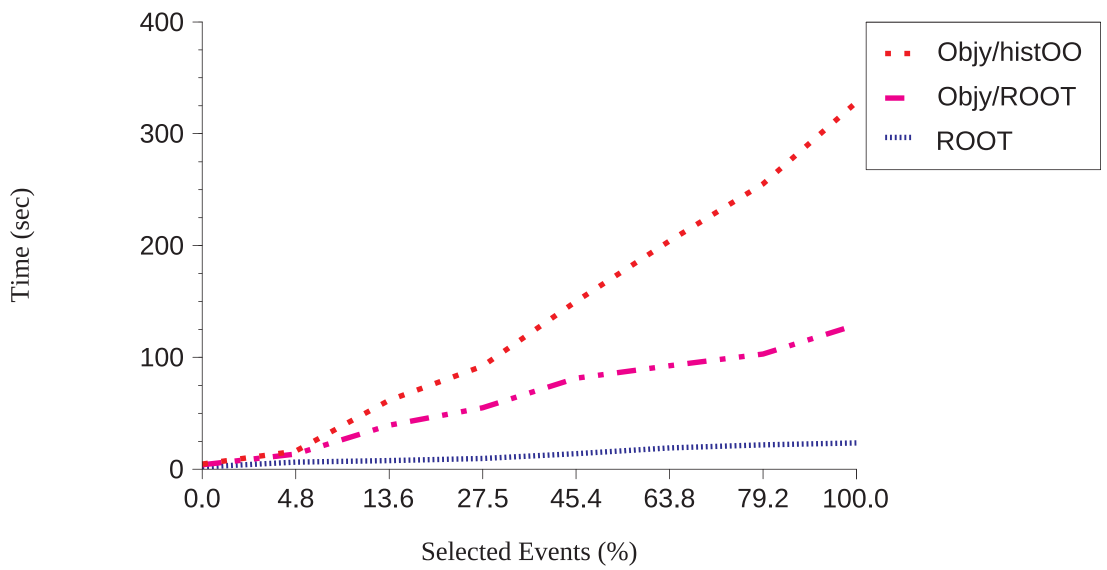

Figure 2: Database + histogramming performance

Histogramming Performance

The following table and figure show the histogramming performance of the histOOgram and ROOT packages[2]. The results are obtained by subtracting the database traversal overhead obtained in the first test from the results of the previous test.

Entries	histOOgram (in seconds)	ROOT (in seconds)
10000	1.0	0.1
186650	2.9	0.3
564738	22.7	0.4
1224797	39.3	1.5
2168384	71.1	1.7
3225376	115.6	2.9
4186002	156.9	3.8
5691244	206.0	5.2
Total time	615.5	15.9

Table 3: Histogramming performance

Here we see that the ROOT `TH1F` class takes about 0.9 µs per fill operation, while the histOOgram `Histo1D` class takes about 36 µs. We've only tested the persistent version of the histOOgram classes so we don't know if the overhead is due to a bad filling algorithm or due to the Objectivity interface. ROOT histograms are always persistent capable. In any case non-persistent histograms make not much sense, so testing them is only of academic interest. Note the logarithmic scale on Figure 3 below.

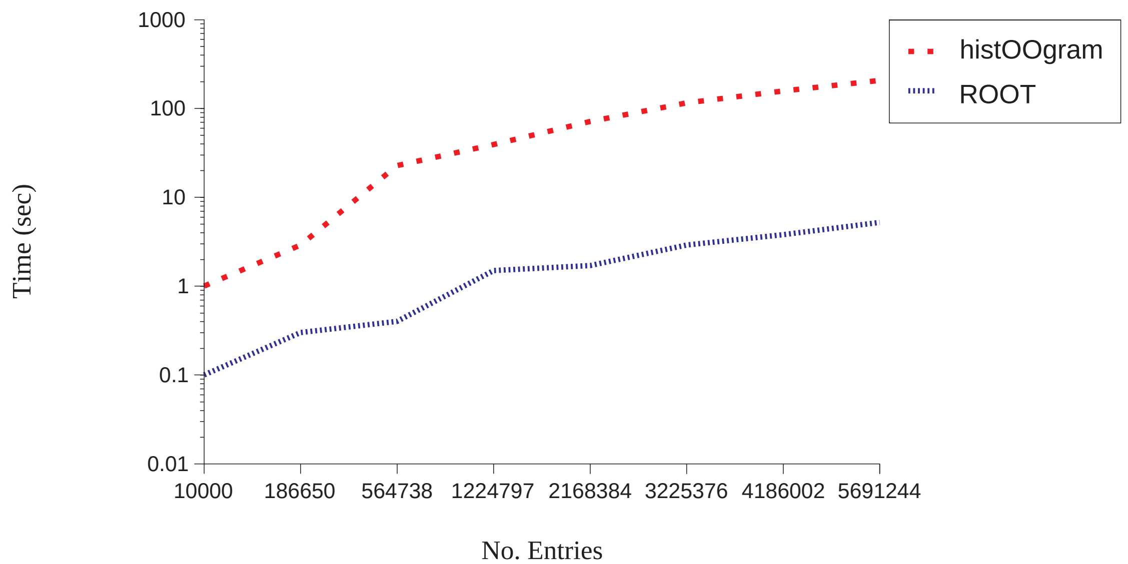

Figure 3: Histogramming performance

[2] In other benchmarks it has been shown that ROOT histogramming is typically between 1.5 to 2 times faster than HBOOK.

Another item to note is the size of the produced histogram database files. The ROOT file containing the 3 histograms is 1700 bytes while the Objectivity file is 137216 bytes.

Conclusions

The results presented in this note clearly show that a system specifically designed and built for data analysis and data mining far outperforms a solution based on generic components badly tuned for the tasks at hand. The Objectivity OODBMS might be a good system for applications in which multiple distributed clients need to access and update a large centrally managed database (e.g. financial trading system). However, it is not specifically designed for efficient read-only database access to parts, or even single attributes, of objects. Objectivity's proposed usage via a few central AMS (multi-threaded access) servers will only make this behaviour worse, since these servers will form a huge bottleneck through which all data must pass.

During this test we have not paid any attention to the feature set of the histOO-gram package, however it clearly needs still a lot of work to make it a competitive solution in terms of performance.

Further it turned out to be absolutely trivial to use ROOT's histogramming facilities in an application which uses Objectivity/DB as event store.

Finally we would like to remark that the Objectivity test program (as shown in Appendix A), including the example event loop, was kindly provided by Dirk Duellmann of the LHC++/RD45 team.

Acknowledgments

The authors would like to thank Predrag Buncic for making sure that shift49 was empty during the running of the benchmarks.

Bibliography

[1] René Brun, Nenad Buncic, Valery Fine & Fons Rademakers, *The ROOT System*. See also http://root.cern.ch/.

[2] Objectivity Inc., Objectivity/DB. See also http://www.objectivity.com/.

[3] CERN Program Library, LHC++. See also http://wwwinfo.cern.ch/asd/lhc++/index.html.

[4] CERN Program Library, HBOOK histogramming package.

Colophon

This paper was produced using the Applixware Office suite v4.3 running on a Linux PC.

Appendix A

The Objectivity/DB based test program including calls to the LHC++ histOOgram package. Note that although Objectivity/DB supports the ODMG OODBMS "standard" API it has been hidden behind a layer called HepODBMS provided by the LHC++ group.

```
#include "iostream.h"
#include "AtlasProd.h"
#include "histo1D.h"
#include "HepODBMS/tagdb/HepTagDbApplication.h"
#include "HepODBMS/tagdb/HepExplorableGenericTags.h"
#include "HepODBMS/tagdb/HepExplorableAttribute.h"


class myAtlasApplication : public HepTagDbApplication {
private :
   void readGenericTags(int npart, int hist);
public:
   myAtlasApplication() {};
   virtual int clusteringInit();
   void run(int npart, int hist);
};

void myAtlasApplication::readGenericTags(int npart, int hist)
{
   HepRef(AtlasProdEvent) evt;
   HepRef(AtlasProdTag) theRealTag;

   // Retrieve the collection of real tags
   HepExplorable *expCollect = HepExplorable::findExplorable("AtlasProdTag");
   if (!expCollect)
      fatal("could not find AtlasProdTag");

   // Open histogram output file and create histograms
   HepDatabaseRef  histoDb = db("HistoDB");
   HepRef(Histo1D) pmass = new(histoDb) Histo1D("Particle Mass", 100, 0., 15.);
   HepRef(Histo1D) tmass = new(histoDb) Histo1D("Total Mass of Event",
                                                100, 0., 600.);
   HepRef(Histo1D) hpart = new(histoDb) Histo1D("Number of particles",
                                                100, 0., 1000.);

   // Start main loop over all events
   float mass, totmass;
   int cnt = 0, sel = 0;
   for (int rc = expCollect->start(); rc != 0 ; rc = expCollect->next()) {

      expCollect->getCurrentObject(theRealTag);

      if (npart == -1 || theRealTag->Npart < npart) {
         evt = theRealTag->getEvent();
         totmass = 0;
         for (int i = 0; i < theRealTag->Npart; i++) {
            mass = evt->particles[i].mass;
            if (hist) pmass->fill(mass);
            totmass += mass;
         }
         if (hist) tmass->fill(totmass);
         sel++;
      }
      if (hist) hpart->fill(theRealTag->Npart);

      cnt++;
      if (!(cnt % 100))
         printf("Processing event %d\n", cnt);
   }

   printf("%f percent of events accepted (%d of %d)\n",
```

```
         sel/(float)cnt*100., sel, cnt);

   printf("number of entries in pmass %d\n", pmass->get_entries_number());
   printf("number of entries in tmass %d\n", tmass->get_entries_number());
   printf("number of entries in hpart %d\n", hpart->get_entries_number());
}

int myAtlasApplication::clusteringInit()
{
   // Call base class first
   if (HepTagDbApplication::clusteringInit()) {
      AtlasProdEvent::clustering = HepEvent::clustering;
      AtlasProdTag::clustering   = HepEventTag::clustering;
   }
   return 1;
}

void myAtlasApplication::run(int npart, int hist)
{
   Init();  // Session init & clustering hints
   if (hist)
      startUpdate();
   else
      startRead();
   readGenericTags(npart, hist);
   commit();
}

int main(int argc, char *argv[])
{
   // By default (-1) calculates totmass for all events,
   // otherwise only for those events that have <npart particles
   int npart = -1;
   if (argc >= 2)
      npart = atoi(argv[1]);

   // By default (1) histogram, to not histogram set to 0
   int hist = 1;
   if (argc == 3)
      hist = atoi(argv[2]);

   myAtlasApplication mApp;

   mApp.run(npart, hist);

   return 0;
}
```

In the above program we see that the object model is broken due to the usage of the, so called, tags. Formally `Npart` is a data member of the `Event` class and should be accessible as: `evt->Npart`. However, now we have: `theRealTag->Npart`.

To use ROOT for histogramming just add the ROOT headers (as in Appendix B) and replace the histOOgram `Histo1D`'s by ROOT `TH1F` objects. To compile and link add `-I$(ROOTSYS)/include` to the compile statement and link with `$(ROOTLIBS)` as shown in `$ROOTSYS/test/Makefile` (which comes with every ROOT distribution).

The header file `AtlasProd.ddl` and the implementation of the `Event` and `Particle` classes is not shown. They are available on the web from URL: http://root.cern.ch/root/Publications.html.

Appendix B

The ROOT based test program.

```
#include <stdlib.h>

#include <TROOT.h>
#include <TFile.h>
#include <TTree.h>
#include <TH1.h>

#include "AtlasProd.h"


void totalmass(int npart = -1, int histo = 1)
{
   // Connect ROOT event database
   TFile db("/shift/shd09/data10/vp/users/rdm/rtdb/event.root");

   // Get TTree container from DB and setup clustering hints
   TTree *h1 = (TTree*)db.Get("Event");
   AtlasProdEvent *event = new AtlasProdEvent;

   h1->SetBranchAddress("event", &event);

   h1->SetBranchStatus("*", 0);               // disable all branches
   h1->SetBranchStatus("Npart", 1);           // activate named branch
   h1->SetBranchStatus("particles.mass", 1);  // activate named branch

   TBranch *bNpart = (TBranch *) h1->GetBranch("Npart");

   // Open histogram output file and create histograms
   TFile hfile("atlas.root", "recreate", "Atlas ROOT Histograms");

   TH1F pmass("pmass", "Particle Mass", 100, 0., 15.);
   TH1F tmass("tmass", "Total Mass of Event", 100, 0., 600.);
   TH1F hpart("hpart", "Number of particles", 100, 0., 1000.);

   // Start main loop over all events
   Float_t mass, totmass;
   int cnt = 0, sel = 0;
   Int_t nentries = (int)h1->GetEntries();
   for (int i = 0; i < nentries; i++) {

      if (npart != -1)
         bNpart->GetEvent(i);

      if (npart == -1 || event->Npart < npart) {
         h1->GetEvent(i);
         totmass = 0;
         for (int j = 0; j < event->Npart; j++) {
            mass = event->GetParticle(j)->mass;
            if (histo) pmass.Fill(mass);
            totmass += mass;
         }
         if (histo) tmass.Fill(totmass);
         sel++;
      }
      if (histo) hpart.Fill(event->Npart);

      cnt++;
      if (!(cnt % 100))
         printf("Processing event %d\n", cnt);
   }

   printf("%f percent of events accepted (%d of %d)\n",
         sel/(float)cnt*100., sel, cnt);
```

```
   printf("number of entries in pmass %f\n", pmass.GetEntries());
   printf("number of entries in tmass %f\n", tmass.GetEntries());
   printf("number of entries in hpart %f\n", hpart.GetEntries());

   // Write histograms to database
   if (histo) hfile.Write();
}


int main(int argc, char *argv[])
{
   // Initialize ROOT system
   TROOT root("AtlasProd", "AtlasProd test");

   // By default (-1) calculates totmass for all events,
   // otherwise only for those events that have <npart particles
   int npart = -1;
   if (argc >= 2)
      npart = atoi(argv[1]);

   // By default (1) histogram, to not histogram set to 0
   int histo = 1;
   if (argc == 3)
      histo = atoi(argv[2]);

   totalmass(npart, histo);

   return 0;
}
```

The header file `AtlasProd.h` and the implementation of the `Event` and `Particle` classes is not shown. They are available on the web from URL: http://root.cern.ch/root/Publications.html.