Redesign of TGeo for concurrent particle transport

ROOT Users Workshop, Saas-Fee 2013

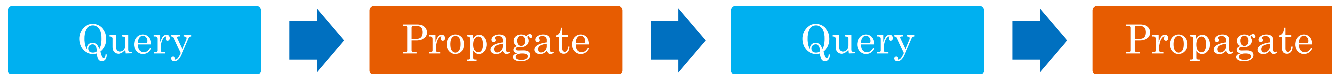
Andrei Gheata

WHY "PARALLELIZING" GEOMETRY ?

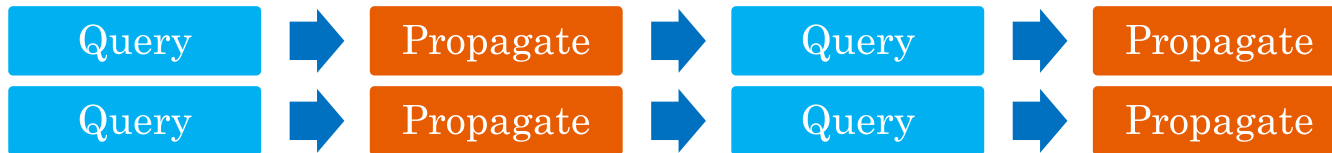
- Key component in many HEP applications
  - Simulation MC, reconstruction, event displays, DB's, …
- Wrapper to 3D or material information
  - Positions, sizes, matrices, alignment, densities, …
  - Optimizing this use case is application responsibility
    - E.g. event display + geometry
- Some of the HEP applications use directly geometry functionality, namely navigation
  - Namely transport MC and tracking code
  - As these applications work or will work concurrently, geometry has to follow…

WHAT KIND OF PARALLELISM ?

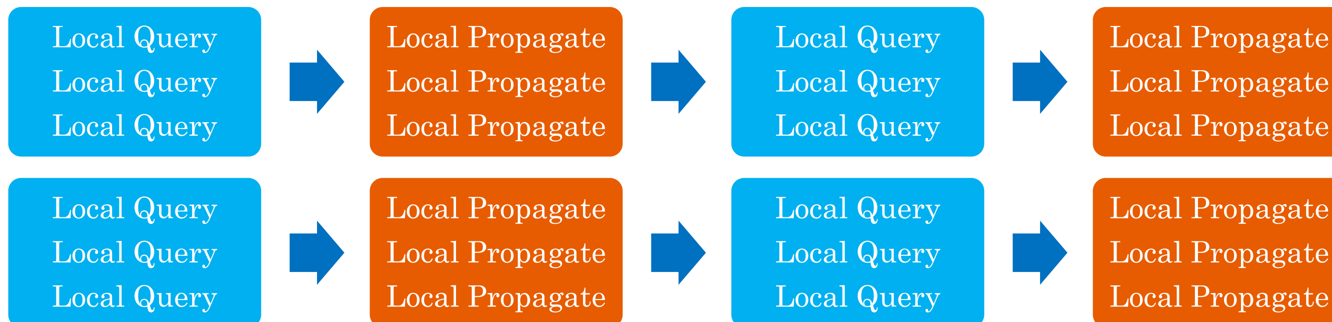
- Sequential usage:



- Parallel transport of events or tracks:



- Parallel local vector transport:



How to implement it?

- Thread safety
  - Achieved and covered in this presentation
- Navigation is the concurrent part – provide a parallelism model here
  - Task-based parallelism (e.g. different tracks/events to different threads)
- Preserve or improve locality
  - Hierarchical navigation, difficult to factorize WITHIN a query
  - Context cannot be stored in the data structures -> propagate with query
- Further finer grain concurrency – solids
  - Main CPU consumer in geometry
  - GPU kernels, other coprocessors
    - Context exchange latency can be a limiting factor – to be tested
- Vectorization – ideal for low level computation
  - Requires vectorized navigation methods AND a client sending vector queries
  - Further discussed in next slides

Geometry data structures

- ROOT geometry was NOT thread safe by design
  - State backed up in common geometry structures (optimization purposes)
- Many methods, including simple getters, were not thread safe
- The context dependent content not clearly separated from the ***const***

```
class TGeoPatternFinder : public TObject
{
...
   Double_t         fStep;          // division step length
   Double_t         fStart;         // starting point on divided axis
   Double_t         fEnd;           // ending point
   Int_t            fCurrent;       // current division element
   Int_t            fNdivisions;    // number of divisions
   Int_t            fDivIndex;      // index of first div. node
   TGeoMatrix      *fMatrix;        // generic matrix
   TGeoVolume      *fVolume;        // volume to which applies
   Int_t            fNextIndex;     //! index of next node
```

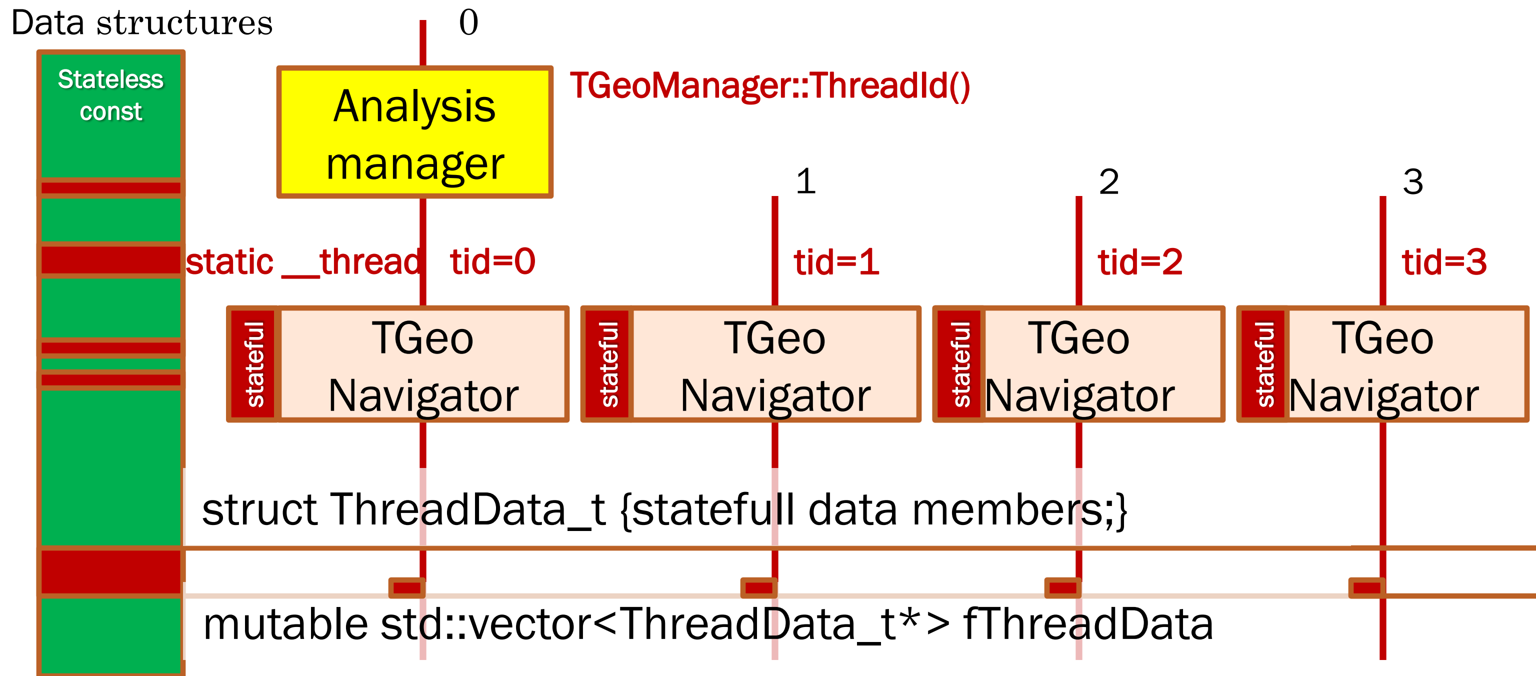
RE-DESIGN STRATEGY

- Thread safety without sacrificing existing optimizations
- Step 1: Split out the navigation part from the geometry manager
  - Most data structures here depend on the state
  - Different calling threads will work with different navigators
- Step 2: Spot all thread unsafe data members and methods within the structural geometry objects and protect them
  - Shapes and optimization structures
  - Convert object->data into object->data[thread_id]
- Step 3: Rip out all context-dependent data from structural objects to keep a compact ***const*** access geometry core
  - Whenever possible, percolate the state in the calling sequence

PROBLEMS ALONG THE WAY

- Separating navigation out of the manager was a tedious process
  - Keeping a large existing API functional
- Spotting the thread-unsafe objects was not obvious
  - Practically all work done by Matevz Tadel (thanks!)
- Changing calling patterns was sometimes impossible, resources needed to be locked
  - First approach suffered a lot from Amdahl law
- Many calls to get the thread Id needed, while there was no implementation of TLS in ROOT
  - *__thread* not supported everywhere

IMPLEMENTATION



- Thread data pre-alocated via *TGeoManager::SetMaxThreads()*
- User threads have to ask for a navigator via *TGeoManager::CreateNavigator()*
- Internal access to a stateful data member goes via:
  - fStatefull ↔ GetThreadData(tid)->fData
  - Non-API methods use percolation: Method(data) ↔ Method(data, &context)

USAGE

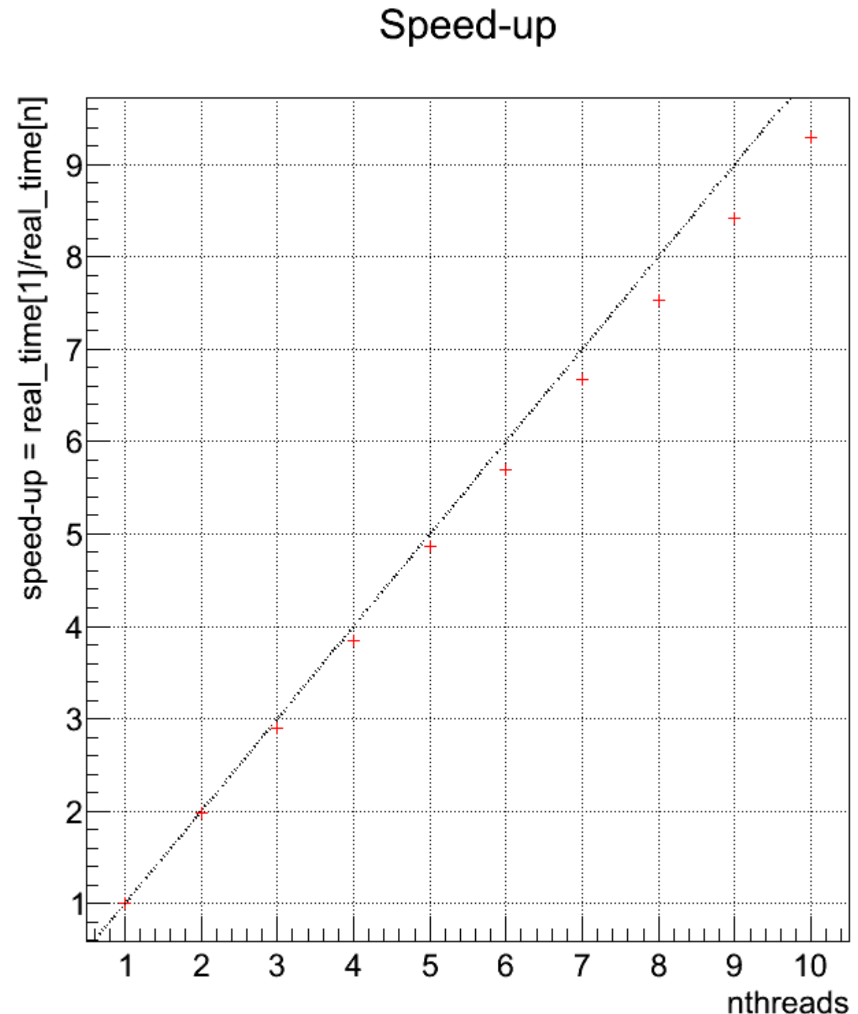
```
//_______________________________________________
MyTransport::PropagateTracks(tracks)
{
// User transport  method called by the main thread
   gGeoManager->SetMaxThreads(N);  // mandatory
   SpawnNavigationThreads(N, my_navigation_thread, tracks)

      void *my_navigation_thread(void  *arg) {
      // Navigation method to be spawned as thread
         TGeoNavigator *nav = gGeoManager->GetCurrentNavigator();
          if (!nav) nav = gGeoManager->AddNavigator();
          int  tid = nav->GetThreadId();      // or   TGeoManager::ThreadId()
          PropagateTracks(subset(tid,tracks));
          return 0;
      }

JoinNavigationThreads();
}
```

SCALABILITY

- Good scalability with rather small Amdahl effects (~0.7 % sequential
  - No lock on memory resources however !
  - Work balancing is not perfect in the test
- Small overheads due to several hidden effects
  - Context switches, false cache sharing (?), pthread calls
  - May need to re-organize context data per thread rather than per object



VECTORIZING THE GEOMETRY

- Assuming the work can be organized such that:
  - Several particles query the same solid (vector navigator)
    - Difficult to achieve, requires vectorized transport
  - A single particle queries several solids of the same type (content navigator)
    - Requires modifying the current navigation optimizers per volume, as well as a "decent" volume content
- What could be the gain?
  - Achieving sizeable speedup factors via SIMD features built into all modern processors
  - Starting the exercise is needed in order to assess how much

TOY EXAMPLE

```
TGeoBBox::Contains(Double_t *point) {
  if (fabs(point[0])>fDX) return false;
  if (fabs(point[1])>fDY) return false;
  if (fabs(point[2])>fDZ) return false;
  return true;
}
```

10.96 sec

```
void ContainsNonVect(Int_t n, point_t *a, bool *inside) {
  int i;
  for (i=0; i<n; i++) {
    inside[i] = (fabs(a[i].x[0]<=fDX) & (fabs(a[i].x[1]<=fDY) & (fabs(a[i].x[2]<=fDZ)
}
```

5.4 sec (x2 original)

```
void ContainsVect(Int_t n, point_t * __restrict__ a, bool *inside) {
  int i;
  for (i=0; i<n; i++) {
    double *temp = a[i].x;
    inside[i] = (fabs(*temp)<=fDX) & (fabs(*(temp+1))<=fDY) & (fabs(*(temp+2))<=fDZ)
}
```

3.74 sec (x3 original)
can expect max . x2 non-vect

Overview

- Thread safety for geometry achieved, introducing 1-2% overhead compared to the initial version
  - Additional ThreadId() and GetThreadData() calls
  - Fast and portable thread ID retrieval implemented via ***ThreadLocalStorage.h***
    - ***__thread*** for Linux/AIX/MACOSX_clang
    - ***__declspec(thread)*** on WIN
    - ***pthread_(set/get)specific*** for SOLARIS, MACOSX
- Parallel navigation has to be enabled:
  - gGeoManager->SetMaxThreads(N)
  - One navigator: per thread: gGeoManager->AddNavigator()
- No locks, very good scalability
  - Small overhead to be adressed by reshuffling context data

NEXT STEPS

- Geometry navigation is not a simple parallel problem
- Re-shuffling of transport algorithms will be needed
  - Propagating vectors from top to bottom
    - When available from the tracking code…
    - Re-think algorithms for solids from this perspective
  - Factorizing loops within a volume
    - Low level optimization at the level of voxels
    - Data flow model and locality to be thought over
      - Minimizing latencies and cache misses when using low-level computational units (GPU)
  - Vectorizing the geometry can potentially gain an important factor
- The time scale is few years, but the work has to start now