DREAMtech - DTK Distributed

Nicolas Niclausse / Thibaud Kloczko / Julien Wintz

Plan

- Contexte & Motivations
- Management des ressources
- Application distribuées
- Conteneurs distribués

Contexte

- ADT DTK (depuis 2012)
  - méta plateforme, pour développer des plateformes modulaires (plugins)
  - multi Plateforme, C++, Qt
  - num3sis, axel, medInria, enas, carbonQuant, pib, inalgae, bolis, winpos
- layers DTK:
  - dtkCore
  - dtkComposer
  - dtkPlot
  - dtkLinearAlgebraSparse
  - ***dtkDistributed***

dtkDistributed: Motivations

- Faciliter l'utilisation des ressources de calcul dans le code
  - indépendances vis à vis des gestionnaires de ressources (oar, torque, accès direct)
  - déploiement automatique
    - ssh (tunnelling si besoin), MPI_comm_spawn
- Simplifier l'écriture d'applications distribuées (API de haut niveau)
- Faciliter le prototypage
  - pas de dépendances en dur sur des bibliothèques tierces (MPI)
  - implémentation par défaut multi plateforme (communicateur qthread)
- Faciliter les intéractions calcul / visualisation

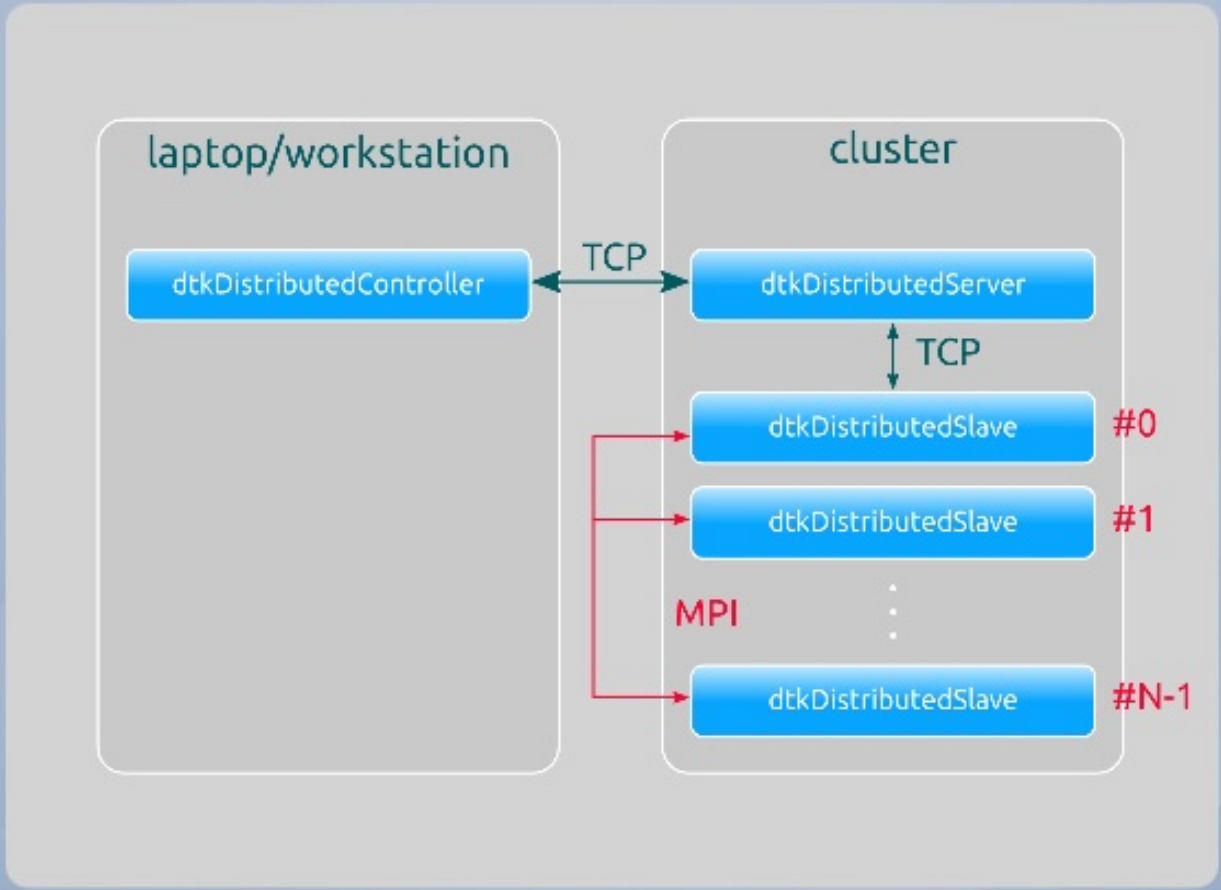
dtkDistributed: Historique

- fin 2011: premier prototype du resource manager
- 2012: communicateur v0, intégration dans le composer dtk
- fin 2012: visu intéractive de simulation distante dans le composer num3sis
- 2013: refonte du communicator sous forme de plugins
- mi 2014: vecteur et graphe distribués
- 2015: intégration dans le layer algèbre sparse

Management de ressources via dtkDistributed

- 3 composants:
  - 1 contrôleur: **dtkDistributedController** (sur la machine cliente (laptop par ex.))
  - 1 serveur **dtkDistributedServer** (sur la frontale du cluster), qui instancie un ressource manager
    - OAR
    - Torque
    - SSH
  - N slaves **dtkDistributedSlave** (sur un(des) noeud(s) d'un cluster)

Management de ressources via dtkDistributed



Application distribuées

- Communicator
- Messages
- Policy, Settings
- Distributed application

Communicator

- dtkDistributedCommunicator
  - interface pour le calcul parallèle
  - paradigme de programmation inspiré de MPI
  - inclus des primitives de:
    - déploiement
    - communication
    - synchronisation

Communicator

- implémenté sour forme de plugins
  - qthread (par défaut)
  - MPI
  - MPI3
- primitives de communication
  - send/receive/ireceive/broadcast/reduce
  - barrier/wait
- buffers distribués
  - get/put/accumulate

Communicator: sérialisation

- primitives de communications compatibles avec:
  - tableau de type simple
  - QVariant
- sérialisation via le QVariant:

```
Q_DECLARE_METATYPE(monType)
qRegisterMetaTypeStreamOperators<monType>("monType");
qRegisterMetaTypeStreamOperators<monType*>("monType*");

// handle the pointer version of qdatastream operators << and >>
#include <dtkMeta/dtkMeta.h>;

QDataStream& operator<<(QDataStream& s, const monType& data);
QDataStream& operator>>(QDataStream& s, monType& data);
```

Communicator: sérialisation

```
 1:  // rank 0 is sending each slave it's mesh
 2:
 3:  if (d->communicator->rank() == 0) {
 4:    dtkContainerVector<numMesh*> *meshes = d->partitioner->meshes();
 5:    for(qlonglong i = 1; i < meshes->count(); ++i) {
 6:       QVariant v = QVariant::fromValue(meshes->at(i));
 7:       d->communicator->send(v, i, tagSend);
 8:       delete meshes->at(i);
 9:    }
10:  } else {
11:    // each slave received it's mesh
12:    QVariant v;
13:    d->communicator->receive(v, 0, tagSend);
14:    d->mesh = v.value<numMesh*>();
15:  }
```

Communicator: sérialisation, vraie version

```
 1:  // rank 0 is sending each slave it's mesh
 2:
 3:  if (d->communicator->rank() == 0) {
 4:    dtkContainerVector<numMesh*> *meshes = d->partitioner->meshes();
 5:    for(qlonglong i = 1; i < meshes->count(); ++i) {
 6:       d->communicator->send(meshes->at(i), i, tagSend);
 7:       delete meshes->at(i);
 8:    }
 9:  } else {
10:    // each slave received it's mesh
11:    d->communicator->receive(d->mesh, 0, tagSend);
12:  }
13:
```

Communicator: spawn

- déploiement du communicateur: `spawn()`
  - *qthread* : instanciation d'une pool de thread (QThreadPool)
  - *mpi* : `MPI_Comm_Spawn`
    - Pas besoin de mpirun (avec openmpi)
- exécution du code parallèle: `exec(QRunnable)`
  - QRunnable: méthode virtuelle `run()` à implémenter

Messages

- échange de messages entre contrôleur, serveur et slave
- protocole basé sur JSON et du pseudo HTTP
- interface C++: `dtkDistributedMessage`
  - Plusieurs méthodes:
    - STATUS: demande d'état du cluster
    - OKSTATUS: état du cluster
    - NEWJOB: création d'un job
    - DELJOB: suppression d'un job
    - DATA: échange de données (QVariant)
    - ...

Policy & Settings

- policy:
  - permet de choisir l'implémentation (qthread, mpi, mpi3)
  - permet de spécifier les machines sur lequel l'application tourne
    - s'interface avec Torque, OAR
    - variable d'environnement DTK_NUM_PROCS
- Settings:
  - fichier ini
    - défini vers le plugins path
    - autres options ...

dtkDistributedApplication

- simplifie l'invocation d'une application
  - configure automatiquement la policy
  - initialise le plugin manager du communicator
- options en ligne de commande
  - nombre de procs, policy, etc.
  - ajout d'option spécifiques.
- déploiement de l'application (spawn)
- exécution des taches de calcul (exec)

```
dtkDistributed::create(argc,argv);

QCommandLineParser *parser=app->parser();

app.initialize();

app.spawn();

app.exec(work);

app.unspawn();
```

Distributed application

```
# myappli --help
Usage: myappli [options]
Lorem ipsum dolor sit amet, consectetur adipiscing elit
Options:
  --myoption <foo|bar>                              myappli option(s)

  --policy <qthread|mpi|mpi3>                       dtkDistributed policy
                                                    (default is qthread)
  --np <int>                                        number of processes
  --hosts <hostname>                                hosts (multiple hosts can be specified)
  -h, --help                                        Displays this help.
  -v, --version                                     Displays version information.
  --settings <filename>                             settings file
  --verbose                                         verbose plugin initialization
  --nw, --no-window                                 non GUI application (no window)
  --loglevel <trace|debug|info|warn|error|fatal>    log level used by dtkLog
                                                    (default is info)
  --logfile <filename | console>                    log file used by dtkLog; default is:
                                                    /home/nniclaus/.local/share/inria/dtk
                                                    DistributedSlave/dtkDistributedSlave.log
```

Example de slave

```
 1:  int main(int argc, char **argv) {
 2:     dtkDistributedAbstractApplication *app=dtkDistributed::create(argc,argv);
 3:     QCommandLineParser *parser = app->parser();
 4:     parser->setApplicationDescription("DTK distributed slave application.");
 5:     QCommandLineOption serverOption("server", "Server URL", "URL");
 6:     parser->addOption(serverOption);
 7:     app->initialize();
 8:     if (!parser->isSet(serverOption)) {
 9:       qCritical() << "Error: no server set ! Use --server <url> " ;
10:       return 1;
11:     }
12:     slaveWork work;
13:     work.server = parser->value(serverOption);
14:     app->spawn();
15:     app->exec(&work);
16:     app->unspawn();
17:     return 0;
18:  }
```

Example de slave (suite)

```
 1:  class slaveWork : public QRunnable {
 2:   public:
 3:     QString server;
 4:   public:
 5:    void run(void) {
 6:       dtkDistributedCommunicator *c=dtkDistributed::communicator::instance();
 7:       dtkDistributedSlave slave;
 8:       slave.connectFromJob(server);
 9:       QThread::sleep(5);
10:       if (c->rank() == 0) {
11:           QString s = QString::number(c->size());
12:           QString hello = "I'm the master slave, we are "+s+" slaves";
13:           QVariant v(hello);
14:           dtkDistributedMessage m(dtkDistributedMessage::DATA,slave.jobId(),
15:                                   dtkDistributedMessage::CONTROLLER_RANK, v);
16:           m.send(slave.socket());
17:       }
18:       QThread::sleep(5); slave.disconnectFromJob(server); }};
```

Examples d'application avec contrôleur/slave

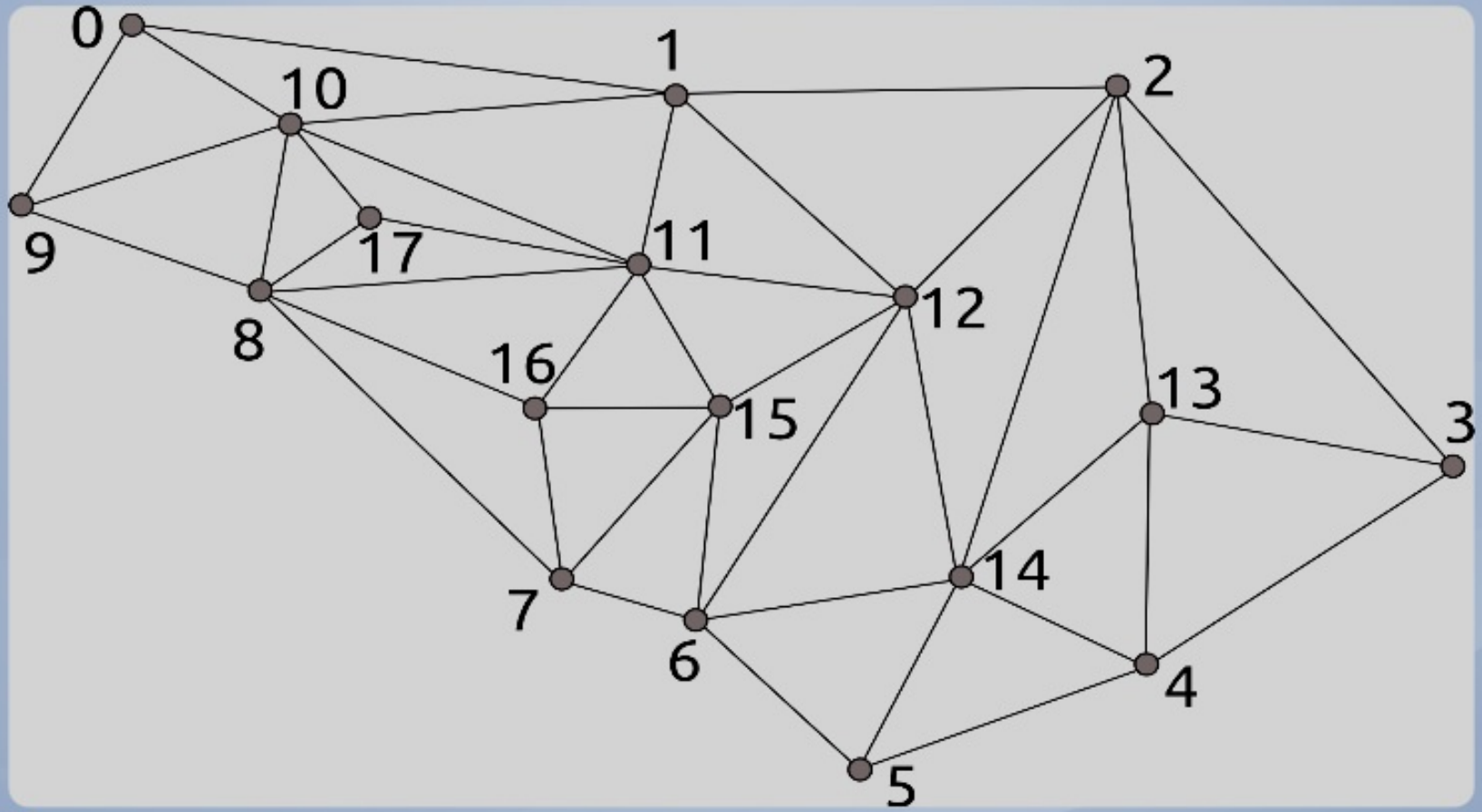
- Dashboard QML
  - Objet 'dtkDistributedController' instancié en QML
  - Serveur sur la frontale choisie
  - Lance un job Slave via le serveur
  - Affiche les données envoyées par le slave

Contrôleur intégré dans numComposer

Conteneurs distribués

- étude d'un cas a priori simple
- état de l'art
- organisation des classes
- dtkDistributedArray
- dtkDistributedGraphTopology
- retour sur le cas initial.

Calcul de maximum sur un graphe

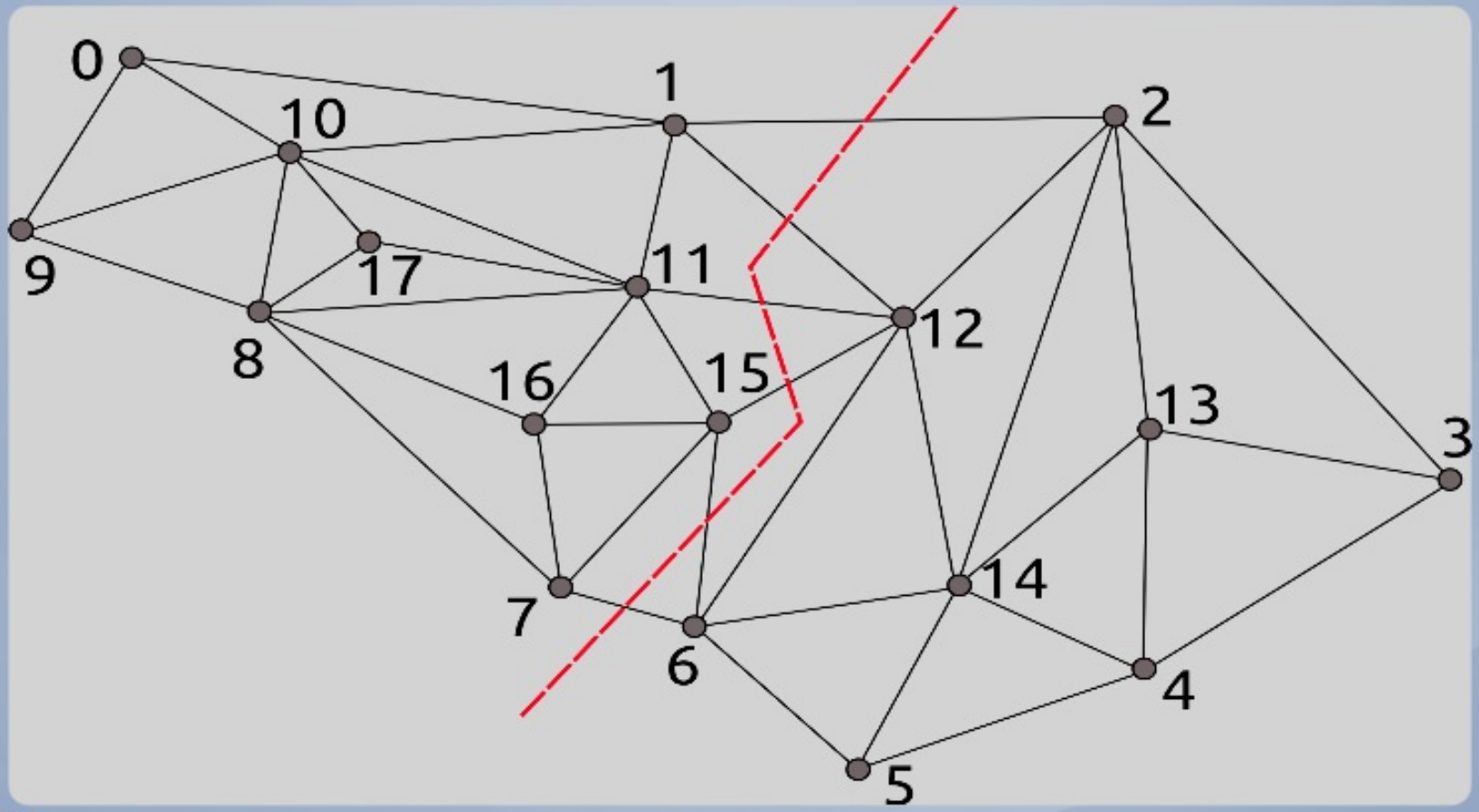


Calcul de maximum sur un graphe

```
Foreach vertex in graph {
   Foreach neighbour of vertex {
       max_values[vertex_id] = Max(max_values[vertex_id], values[neighbour_id]);
   }
}
```

- Structure de graphe simple
- Algorithme simple

Calcul de maximum sur un graphe distribué



Calcul de maximum sur un graphe distribué

```
// Local Computation
Foreach vertex in subGraph {
    Foreach neighbour of vertex {
        max_values[vertex_id] = Max(max_values[vertex_id], values[neighbour_id]);
    }

// Boundary computation
Foreach boundary in subGraph {
    Foreach vertex of Boundary {
        local_max_values[local_id] = max_values[vertex_id];
    }
    Send(local_max_values);

    Receive(extern_max_values);
    Foreach vertex of Boundary {
        max_values[vertex_id] = Max(max_values[vertex_id], extern_max_values[ext_id]);
    }
}
```

- Complexité accrue en terme de structure et d'algorithme

Calcul du maximum sur les voisins des voisins

- :-/
- Refaire les structures
- Repenser l'algorithmie
- ...
- Tout compte fait, on va s'en passer...

Bibliothèques existantes

- Parallel Standard Library (PSTL) -> plus maintenu
- Intel Threadings Building Blocks (TBB) -> seulement multithread
- Parallel Object-Oriented Methods and Applications (POOMA) -> pas de structure de graphe
- Standard Template Adaptive Parallel Library (STAPL) -> ni sources ni binaires
- Fortran Coarray (OpenCorrays) -> code C qui génère du Fortran!

dtkDistributed Containers

Containers

- dtkDistributedArray<T>
- dtkDistributedGraphTopology

Distribution Environment

- dtkDistributedCommunicator
- dtkDistributedContainer

Container Accessor

- dtkDistributedIterator<T>
- dtkDistributedNavigator<T>

Memory Management

- dtkDistributedMapper
- dtkDistributedBufferManager
- dtkDistributedArrayCache<T>

dtkDistributedArray : API

```
 1:  template < typename T > class dtkDistributedArray :
 2:                                        public dtkDistributedContainer {
 3:  public:
 4:    dtkDistributedArray(const qlonglong& size, dtkDistributedMapper *mapper);
 5:   ~dtkDistributedArray(void);
 6:  public:
 7:    bool empty(void) const;
 8:    qlonglong size(void) const;
 9:    void fill(const T& value);
10:    void setAt(const qlonglong& index, const T& value);
11:    T at(const qlonglong& index) const;
12:    T operator[](const qlonglong& index) const;
13:  public:
14:    iterator  begin(void);
15:    iterator    end(void);
16:    const T *data(void) const;
17:          T *data(void);
```

dtkDistributedArray : Sequential initialization

```
 1:  qlonglong N = 16257;
 2:  dtkDistributedCommunicator *comm = dtkDistributed::communicator::instance();
 3:  qlonglong *input = new qlonglong[N];
 4:  for (int i = 0; i < N; ++i) {
 5:      input[i] = i;
 6:  }
 7:  dtkDistributedArray<qlonglong> array(N);
 8:  if (comm->wid() == 0) {
 9:      for (int j = 0; j < N; ++j) {
10:          array.setAt(j, input[j]);
11:      }
12:      for (int j = 0; j < N; ++j) {
13:          if(array.at(j) != input[j]) {
14:              return false;
15:          }
16:      }
17:  }
18:  comm->barrier();
```

dtkDistributedArray : Parallel initialization

```
 1:  qlonglong N = 16257;
 2:  dtkDistributedCommunicator *comm = dtkDistributed::communicator::instance();
 3:  qlonglong *input = new qlonglong[N];
 4:  for (int i = 0; i < N; ++i) {
 5:      input[i] = i;
 6:  }
 7:
 8:  dtkDistributedArray<qlonglong> array(N);
 9:
10:  dtkDistributedArray<qlonglong>::iterator ite = array.begin();
11:  dtkDistributedArray<qlonglong>::iterator end = array.end();
12:
13:  for(int i = 0; ite != end; ++ite, ++i) {
14:      *ite = input[array.mapper()->localToGlobal(i, comm->wid())];
15:  }
16:  comm->barrier();
```

dtkDistributedArray : globale vs locale

```
bool empty(void) const;
qlonglong size(void) const;

void fill(const T& value);

void setAt(const qlonglong& id, const T& val);

T at(const qlonglong& id) const;
T operator[](const qlonglong& id) const;
```

```
iterator  begin(void);
iterator    end(void);

const T *data(void) const;
      T *data(void);
```

dtkDistributedArray : architecture

```
// all processes duplicate the mapper
dtkDistributedCommunicator *comm;
dtkDistributedMapper *mapper;
```

```
// unit process #0
dtkArrayData *data;
dtkDSBufferManager *bm;
dtkDSArrayCache *cache;
```

```
// unit process #1
dtkArrayData *data;
dtkDSBufferManager *bm;
dtkDSArrayCache *cache;
```

```
// unit process #2
dtkArrayData *data;
dtkDSBufferManager *bm;
dtkDSArrayCache *cache;
```

dtkDistributedArray : allocation mémoire

```
 1:  template <typename T> inline void dtkDistributedArray<T>::allocate(void)
 2:  {
 3:      // Communicator creates the buffer manager matching its
 4:      // implementation (Qthread, MPI, MPI3)
 5:
 6:      bm = comm->createBufferManager();
 7:
 8:      // Buffer manager allocates the memory according to size given by
 9:      // the mapper
10:
11:      void *local_buffer = bm->allocate<T>(mapper->localSize());
12:
13:      // This buffer initializes the local container
14:
15:      data = dtkArrayData<T>::fromRawData(buffer, mapper->localSize());
16:  }
```

dtkDistributedArray : accès aux valeurs

```
 1: template <typename T> inline T dtkDistributedArray<T>::at(qlonglong id) const
 2: {
 3:     // Identifies the unit process that has the value
 4:     qint32 owner = mapper->owner(id);
 5:     // Identifies the local position of the value on the owner
 6:     qlonglong pos = id - mapper->firstId(owner);
 7:
 8:     // If the owner is the local process the access is direct
 9:     if (this->wid() == owner) {
10:         return data[pos];
11:     // Else the remote value is asked by the buffer manager
12:     } else {
13:         T val;
14:         bm->get(owner, pos, &val);
15:         return val;
16:     }
17: }
```

Buffer Manager : accès aux valeurs en Qthread

```
 1:  void qthDistributedBufferManager::get(qint32 owner, qlonglong pos,
 2:                                                       void *array,
 3:                                                       qlonglong nelement = 1)
 4:  {
 5:      // Owner id enables to get the right shared buffer
 6:
 7:      char *buffer = d->buffers[owner];
 8:
 9:
10:      // Copy into array is done from pos to pos + nelement
11:
12:      memcpy(array, buffer + pos * d->object_size, d->object_size * nelement);
13:  }
```

- Accès direct au buffer cible

Buffer Manager : accès aux valeurs en MPI

```
 1:  void mpiDistributedBufferManager::get(qint32 owner, qlonglong pos,
 2:                                        void *array, qlonglong nelement = 1)
 3:  {
 4:      // Size of the memory to get
 5:      qlonglong array_size = d->object_size * nelements;
 6:
 7:      // Copy into array from the memory of the owner at position pos
 8:      MPI_Get(array, array_size, MPI_BYTE,
 9:              owner, pos, array_size, MPI_BYTE, d->win);
10:  }
```

- Utilisation de MPI One-Sided Communication
  - permet à un processus d'accéder à un espace d'adresse d'un autre processus sans la participation explicite de celui-ci.
  - permet de simplifier le programme et de coder comme en mémoire partagée

MPI One-Sided Communication

- `MPI_Win_Create`
  - crée une fenêtre pour l'accès par d'autres processus à une zone mémoire
- `MPI_Win_Free`
  - détruit la fenêtre
- `MPI_Get / MPI_Put`
  - lecture/écriture asynchrone sur une zone mémoire exposée par une fenêtre
- Bibliothèques utilisant MPI One-Sided Communication :
  - *OpenCoarrays* pour les Coarrays en Fortran 2008
  - *OpenShmem* implémentation de PGAS (Partitioned Global Address Space)

dtkDistributedGraphTopology : API

```
 1:  class dtkDistributedGraphTopology : public dtkDistributedContainer
 2:  {
 3:  public:
 4:    dtkDistributedArray(qlonglong vertex_count, dtkDistributedMapper *mapper);
 5:   ~dtkDistributedArray(void);
 6:
 7:    void addEdge(qlonglong from, qlonglong to, bool oriented = false);
 8:    void build(void);
 9:
10:    qlonglong vertexCount(void) const;
11:    qlonglong   edgeCount(void) const;
12:
13:    qlonglong neighbourCount(qlonglong vertex_id) const;
14:    Neighbours operator[](qlonglong vertex_id) const;
15:
16:    iterator begin(void) const;
17:    iterator   end(void) const;
18:  };
```

dtkDistributedGraphTopology : Construction

```
 1:  dtkDistributedGraphTopology graph(18);
 2:
 3:  // Add edges into provisionnal local data structure of the graph
 4:  graph.addEdge(0, 1);
 5:  graph.addEdge(1, 2);
 6:  ...
 7:  graph.addEdge(16, 17);
 8:
 9:  // Build the true structures of the graph
10:  graph.build();
```

- Structure type ParCSR mais avec des dtkDistributedArray
  - `Vertex_to_FirstEdge` : index de la première arête de chaque sommet
  - `Edge_to_Vertex` : index du sommet cible de chaque arête

dtkDistributedGraphTopology : API globale

```
 1:  typedef dtkDistributedNavigator<dtkDistributedArray<qlonglong> > Neighbours;
 2:
 3:  // Gets a navigator for neighbours of a vertex
 4:  Neighbours n = graph[i];
 5:  qlonglong n_count = n.size();
 6:
 7:  // Can iterate over all the neighbours of the vertex
 8:  Neighbours::iterator nit  = n.begin();
 9:  Neighbours::iterator nend = n.end();
10:
11:  for(; nit != nend; ++nit) {
12:      // Access to id of each neighbour
13:      qDebug() << *nit;
14:  }
```

- `dtkDistributedNavigator` interroge le graphe via l'API globale.

dtkDistributedGraphTopology : API locale

```
 1:  typedef dtkDistributedIterator<dtkDistributedGraphTopology > iterator;
 2:
 3:  // Iterates over all local vertices
 4:  iterator vit  = graph.begin();
 5:  iterator vend = graph.end();
 6:
 7:  for(; vit != vend; ++vit) {
 8:      // Access to all the neighbours of the local vertex
 9:      Neighbours n = *vit;
10:  }
```

- En combinant `iterator` local et `navigator` global, on peut:
  - accéder aux sommets voisins non-locaux
  - accéder aux voisins des voisins non-locaux
  - accéder aux voisins des voisins non-locaux des voisins non-locaux ;-)

Calcul du maximum sur les voisins des voisins

```
 1:  dtkDistributedGraphTopology::iterator it  = graph.begin();
 2:  dtkDistributedGraphTopology::iterator end = graph.end();
 3:
 4:  for(;it != end; ++it) {
 5:      dtkDistributedGraphTopology::Neighbours n = *it;
 6:      dtkDistributedGraphTopology::Neighbours::iterator nit  = n.begin();
 7:      dtkDistributedGraphTopology::Neighbours::iterator nend = n.end();
 8:
 9:      for(;nit != nend; ++nit) {
10:          dtkDistributedGraphTopology::Neighbours nn = graph[*nit];
11:          dtkDistributedGraphTopology::Neighbours::iterator nnit  = nn.begin();
12:          dtkDistributedGraphTopology::Neighbours::iterator nnend = nn.end();
13:
14:          for(;nnit != nnend; ++nnit) {
15:              max_values[it.id()] = qMax(max_values[it.id()], values[*nnit]);
16:          }
17:      }
18:  }
```

En résumé, ...

- Avec :
  - une structure fondamentale :
    - `dtkDistributedArray`
  - une structure aggrégatrice :
    - `dtkDistributedGraphTopology`
  - et des structures périphériques d'accès :
    - `dtkDistributedIterator`
    - `dtkDistributedNavigator`
- le code utilisateur parallèle s'écrit comme en séquentiel
- YES, WE CAN !

Algèbre Linéaire Creuse

- dtkDistributedSparseMatrix
- dtkLinearSolverSparse

dtkSparseMatrix

- `dtkSparseMatrix`: Abstraction de Sparse Matrix
  - itérateur de lignes:
    `dtkSparseMatrixLine dtkSparseMatrix::begin()/end()`
  - itérateur sur les éléments de lignes:
    `dtkSparseMatrixLineElement dtkSparseMatrixLine::begin()/end()`
- `dtkSparseMatrixData`: implémentation séquentielle
- `dtkDistributedSparseMatrixData`: implémentation parallèle:
  - `dtkDistributedGraphTopology`
  - `dtkDistributedArray` (valeurs)

dtkSparseMatrix

```
 1:  dtkSparseMatrixLine<double> line_it = mat.begin();
 2:  dtkSparseMatrixLine<double> line_end = mat.end();
 3:
 4:  for (; line_it != line_end; ++line_it) {
 5:      dtkSparseMatrixLineElement<double> elt_it  = line_it.begin();
 6:      dtkSparseMatrixLineElement<double> elt_end = line_it.end();
 7:      qlonglong i = line_it.id();
 8:      for(;elt_it != elt_end; ++elt_it) {
 9:          qlonglong j = elt_it.id();
10:          // accès à value via l'iterateur:
11:          value =  (*elt_it);
12:          // modification via i,j (numérotation globale)
13:          mat(i,j) += coef;
14:          // en MPI, '+=' implémenté via MPI_Accumulate()
15:      }
16:  }
17:
```

dtkSparseSolver

- API générique (séquentielle/parallèle)
- implémentations sous forme de plugins
  - dtkSparseSolverJacobi (générique)
  - hypreDistributedSparseSolver
    - utilisation de la dtkDistributedSparseMatrixData
    - accès direct aux buffers locaux
  - à venir: mumps, MaPHyS, ...

dtkSparseSolver<T> API

```
 1:  template < typename T > class dtkSparseSolver : public QRunnable
 2:  {
 3:  public:
 4:                  dtkSparseSolver(void) {;}
 5:      virtual ~dtkSparseSolver(void) {;}
 6:
 7:  public:
 8:    virtual void setMatrix(dtkSparseMatrix<T> *matrix) = 0;
 9:    virtual void setRHSVector(dtkVector<T> *rhs_vector) = 0;
10:    virtual void setSolutionVector(dtkVector<T> *sol_vector) = 0;
11:
12:    virtual void setOptionalParameters(const QHash<QString,int>& parameters);
13:    virtual void setOptionalParameters(const QHash<QString,double>& parameters);
14:
15:  public:
16:      virtual void run(void) = 0;
17:  public:
18:      virtual dtkVector<T> *solutionVector(void) const = 0;
```

dtkSparseSolverJacobi: implémentation générique

```
 1:  while(!stop_criterion) {
 2:      line_it = mat.begin(); line_end = mat.end(); tmp=sol; d->variation=0;
 3:      for (; line_it != line_end; ++line_it) {
 4:          dtkSparseMatrixLineElement<double> elt_it  = line_it.begin();
 5:          dtkSparseMatrixLineElement<double> elt_end = line_it.end();
 6:          neighbour = 0; diag = 0; qlonglong i = line_it.id();
 7:          for(;elt_it != elt_end; ++elt_it) {
 8:              qlonglong j = elt_it.id();
 9:              if (j != i) { neighbour += (*elt_it) * tmp[j];}
10:              else        { diag = (*elt_it); }}
11:          sol[i] = (rhs.at(i) - neighbour) / diag ;}
12:      tmp -= sol;
13:      d->variation = tmp.asum();
14:      if(iteration_count == 0) { variation_ini = d->variation;}
15:      if(iteration_count == d->number_of_iterations ||
16:           d->variation/variation_ini < d->residual_reduction_order)
17:           stop_criterion = true;
18:      ++iteration_count;  sol.clearCache(); tmp.clearCache();}
```

dtkSparseMatrixSolverApp

```
>./bin/dtkSparseMatrixSolverApp --help
Usage: ./bin/dtkSparseMatrixSolverApp [options]
DTK sparse matrix solver.
Options:
  --matrix <file>                                  matrix (matrixmarket format)
  --rhs <file>                                     rhs vector (matrixmarket format)
  --sol <file>                                     solution vector to be written
  --refsol <file>                                  reference solution
  --impl <sequential|distributed>                  implementation
  --solver <jacobi|SGS|...>                        solver implementation.
  --iterations <integer>                           number of iterations
  --rro <double>                                   residual reduction order
  --policy <qthread|mpi|mpi3>                      dtkDistributed policy
  --np <int>                                       number of processes
  --hosts <hostname>                               hosts (multiple hosts can be specified)
  -h, --help                                       Displays this help.
  -v, --version                                    Displays version information.
  --settings <filename>                            settings file
  --verbose                                        verbose plugin initialization
  --nw, --no-window                                non GUI application (no window)
  --loglevel <trace|debug|info|warn|error|fatal>   log level used by dtkLog
  --logfile <filename | console>                   log file used by dtkLog;
```

Max des voisins des voisins

- graphe, noeuds: 10M arcs: 75M
- cluster nef, nœuds Xeon 20 cœurs, infiniband 40G
- openmpi-1.8.5.rc2
  - mpi3 1 thread: 30.0 sec
  - mpi3 2 threads: 15.4 sec
  - mpi3 10 threads: 3.4 sec
  - mpi3 20 threads: 2.0 sec
  - mpi3 2x20 threads: 7.7 sec
  - mpi3 4x20 threads: 13.6 sec
  - mpi3 8x20 threads: 21.5 sec

Jacobi générique vs Hypre

- matrice 13k x 13k, nnz: 365k, jacobi: 1512 iter.
- cluster nef, nœuds Xeon 20 cœurs, infiniband 40G
- openmpi-1.8.5.rc1
  - jacobi sequentiel: 5.3 sec
  - jacobi   20 procs: 1.1 sec
  - jacobi 2x20 procs: 13.4 sec
  - jacobi 10x4 procs: 7.3 sec
  - hypre   1 proc: 0.7 sec
  - hypre   4x4 procs: 0.14 sec

solveur Jacobi, résultats préliminaires

- matrice 2M x 2M, nnz: 138M
- jacobi: 100 iterations
- cluster nef, nœuds Xeon 20 cœurs, infiniband 40G
- mvapich 2.1
  - jacobi   1   proc: 188 sec
  - jacobi 20 procs: 30 sec
  - jacobi 4x20 procs: 101 sec

plugin MPI3: Retour d'expérience

- optimisations en mémoire partagés
  - `MPI_Win_allocate_shared`
  - `MPI_Comm_split_type` avec MPI_COMM_TYPE_SHARED
- communications RMA (One Sided) with Passive target
- implémentations MPI3 encore peu matures:
  - bugs remontés à openmpi, mvapich (corrigés)
  - mais: *An important part of MPI-3 deals with improved semantics and features for MPI RMA to address some of these criticisms and make MPI RMA a portable runtime system that can provide high-performance and feature-rich one-sided communication support to applications.* Jeff Squyres

Perspectives

- ADT SIMON (Jérémie Labroquère et Tristan Cabel)
  - intégration de dtkSparseMatrix et dtkSparseSolver dans numWindSimulator (presque fait)
  - dtkMesh basé sur dtkDistributedGraphTopology
  - refonte de numWindSimulator basé sur dtkMesh distribué
  - plugins MaPHyS
- optimisations
- intégration dans le composer des structures distribuées de haut niveau

Contacts

dtk-team@inria.fr