ANGEWANDTE MATHEMATIK UND INFORMATIK
UNIVERSITÄT ZU KÖLN

Report 99.359

Parallel ABACUS – Implementation

June 1999

Max Böhm

This work has been partially supported by DFG-Projekt Ju 204/4-2, Re 776/5-3.

1. THE PARALLEL MASTER.

The class **ABA_PARMASTER** contains all data and operations specific to the parallel ABACUS.

```
⟨ parmaster.h 1 ⟩ ≡
#ifndef ABA_PARMASTER_H
#define ABA_PARMASTER_H
#include "abacus/abacusroot.h"
#include "abacus/array.h"
#include "abacus/sub.h"
#include "abacus/idmap.h"
#include "abacus/message.h"
#include "abacus/broadcast.h"
#include "abacus/dualbound.h"
  class ABA_MASTER;
  class ABA_NOTIFYSERVER;
  class ABA_SUBSERVER;
  class ABA_BALANCER;
  template⟨class Type⟩ class ABA_IDMAP;
  class ABA_PARMASTER : public ABA_ABACUSROOT {
  public:
    ABA_PARMASTER(ABA_MASTER *master);
    ~ABA_PARMASTER();
    void initializeParameters();
    void setDefaultParameters();
    void printParameters();
    void outputStatistics();
    int hostId() const;
    int hostCount() const;
    bool isHostZero() const;
    const ABA_STRING &hostname(int i) const;
    const ABA_STRING &hostname() const;
    int registerPool(void *pool);
    void unregisterPool(int index);
    void *getPool(int index);
    ABA_IDMAP⟨ABA_SUB⟩ *subIdentificationMap() const;
    void connectService(int port, ABA_MESSAGE &msg, int destId);
    ACE_SOCK_Stream &notifyStream(int i);
    void newHostDualBound(double x);
    void newHostDualBound(ABA_MESSAGE &msg);
    void newPrimalBound(double x);
    void newPrimalBound(ABA_MESSAGE &msg);
    void newOpenSubCount(int n, double best);
    void newOpenSubCount(ABA_MESSAGE &msg);
    bool balance();
    void terminationCheck();
    void terminationCheck(ABA_MESSAGE &msg);
    void startTerminationCheck();
    void incWorkCount();
    void decWorkCount();
    void terminate();
```

```
  int newId(int fatherId);
  void incSubSentCount();
  void incSubReceivedCount();
  void startIdleTime();
  void stopIdleTime();
  void updateIdleTimers(bool first);
  void printId(int id);
private:
  ABA_MASTER *master_;
  int hostId_;
  int hostCount_;
  ABA_STRING myHostname_;
  ABA_ARRAY<ABA_STRING *> hostname_;
  int connectTimeout_;
  ABA_ARRAY<void *> registeredPools_;
  int lastRegisteredPool_;
  ABA_DUALBOUND hostDualBounds_;
  ACE_Thread_Mutex newHostDualBoundMutex_;
  ACE_Thread_Mutex newPrimalBoundMutex_;
  int notifyPort_;
  ACE_SOCK_Stream *notifyStreams_;
  ABA_NOTIFYSERVER *notify_;
  int subserverPort_;
  ABA_SUBSERVER *subserver_;
  int balancerPort_;
  double bestFirstTolerance_;
  ABA_BALANCER *balancer_;
  ABA_ARRAY<int> openSubCount_;
  ABA_DUALBOUND openSubBest_;
  ACE_Thread_Mutex newOpenSubCountMutex_;
  ACE_Thread_Mutex terminationCheckMutex_;
  volatile bool terminationCheckAgain_;
  ACE_Thread_Mutex terminationMutex_;
  volatile bool terminationOk_;
  volatile bool hasTerminated_;
  int workCount_;
  ACE_Thread_Mutex idCounterMutex_;
  int idCounter_;
  int subSentCount_;
  int subReceivedCount_;
  ABA_COWTIMER idleCowTimeFirst_;
  ABA_COWTIMER idleCowTimeMiddle_;
  ABA_COWTIMER idleCowTimeLast_;
};
inline int ABA_PARMASTER::hostId() const
{
  return hostId_;
}
inline int ABA_PARMASTER::hostCount() const
```

```
  {
    return hostCount_;
  }
  inline bool ABA_PARMASTER::isHostZero() const
  {
    return hostId_ ≡ 0;
  }
  inline const ABA_STRING &ABA_PARMASTER::hostname(int i) const
  {
    return *(hostname_[i]);
  }
  inline const ABA_STRING &ABA_PARMASTER::hostname() const
  {
    return *(hostname_[hostId_]);
  }
  inline ACE_SOCK_Stream &ABA_PARMASTER::notifyStream(int i)
  {
    return notifyStreams_[i];
  }
  inline void ABA_PARMASTER::incSubSentCount()
  {
    subSentCount_++;
  }
  inline void ABA_PARMASTER::incSubReceivedCount()
  {
    subReceivedCount_++;
  }
  inline void ABA_PARMASTER::startIdleTime()
  {
    idleCowTimeLast_.start();
  }
  inline void ABA_PARMASTER::stopIdleTime()
  {
    idleCowTimeLast_.stop();
  }
  inline void ABA_PARMASTER::updateIdleTimers(bool first)
  {
    if (first) idleCowTimeFirst_.addCentiSeconds(idleCowTimeLast_.centiSeconds());
    else idleCowTimeMiddle_.addCentiSeconds(idleCowTimeLast_.centiSeconds());
    idleCowTimeLast_.reset();
  }
#endif      /* ¬ABA_PARMASTER_H */
```

2. The member functions are defined in the file `parmaster.cc`.

⟨ `parmaster.cc` 2 ⟩ ≡

```
#include "abacus/parmaster.h"
#include "abacus/master.h"
#include "abacus/pool.h"
#include "abacus/notifyserver.h"
#include "abacus/notification.h"
#include "abacus/subserver.h"
#include "abacus/balancer.h"
#include "abacus/opensub.h"
#include <ace/OS.h>
#include <ace/SOCK_Acceptor.h>
#include <ace/SOCK_Connector.h>     /* test !!! */
#include "abacus/debug.h"
  int debugLevel_ = 0;      /* currently global! */
```

See also sections 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, and 27.

3. The constructor.

Arguments:
master
A pointer to the master object.

```
⟨ parmaster.cc 2 ⟩ +≡
ABA_PARMASTER::ABA_PARMASTER(ABA_MASTER *master):
  master_(master),
  hostId_(0),
  hostCount_(0),
  myHostname_(master),
  hostname_(master,1,0),
  connectTimeout_(10),
  registeredPools_(master,10,0),
  lastRegisteredPool_(-1),
  hostDualBounds_(master),
  notifyStreams_(0),
  notify_(0),
  openSubCount_(master,1,0),
  openSubBest_(master),
  terminationOk_(false),
  hasTerminated_(false),
  workCount_(0),
  idCounter_(1),
  subSentCount_(0),
  subReceivedCount_(0),
  idleCowTimeFirst_(master),
  idleCowTimeMiddle_(master),
  idleCowTimeLast_(master)
  {
    char help[100];
    if (ACE_OS::hostname(help,100) ≡ -1) {
      master→err() ≪ "ABA_PARMASTER::ABA_PARMASTER():␣can't␣determine␣my␣hostname." ≪ endl;
      exit(Fatal);
    }
    myHostname_ = help;
    strcat(help,":");
    master_→out().setPrompt(help);
    master_→err().setPrompt(help);
  }
```

4. The destructor.

```
⟨ parmaster.cc 2 ⟩ +≡
  ABA_PARMASTER::~ABA_PARMASTER()
  {
    if (debug(DEBUG_TERMINATION))
      master_->out() << "DEBUG_TERMINATION: terminating BALANCER..." << endl;
    ACE_Thread_Manager::instance()->cancel_task(balancer_);
    ACE_Thread_Manager::instance()->wait_task(balancer_);
    delete balancer_;
    if (debug(DEBUG_TERMINATION))
      master_->out() << "DEBUG_TERMINATION: terminating SUBSERVER..." << endl;
    ACE_Thread_Manager::instance()->cancel_task(subserver_);
    ACE_Thread_Manager::instance()->wait_task(subserver_);
    delete subserver_;
    if (debug(DEBUG_TERMINATION))
      master_->out() << "DEBUG_TERMINATION: terminating NOTIFICATIONSERVER..." << endl;
    ABA_BROADCAST bcast(master_, true);
    bcast.pack(ABA_NOTIFYSERVER::ShutdownTag);
    bcast.send();
    bcast.receiveReply();      /* receive accumulated reply */
    if (debug(DEBUG_TERMINATION))
      master_->out() << "DEBUG_TERMINATION: shutdown  broadcasted." << endl;
    ACE_Thread_Manager::instance()->wait_task(notify_);
    delete notify_;
    delete notifyStreams_;
  }
```

5. The function *initializeParameters*( ) initializes the parameters specific to the parallel version of ABACUS.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::initializeParameters()
  {
    master_→assignParameter(debugLevel_, "ParallelDebugLevel", 0, INT_MAX, 0);
    master_→assignParameter(connectTimeout_, "ParallelConnectTimeout", 1, INT_MAX, 10);
    master_→assignParameter(hostCount_, "ParallelHostCount", 1, INT_MAX);      /* resize the arrays */
    hostname_.realloc(hostCount_, 0);
    openSubCount_.realloc(hostCount_, 0);
    hostId_ = -1;
    for (int i = 0; i < hostCount_; i++) {
      char help[100];
      sprintf(help, "ParallelHostname_%d", i);
      hostname_[i] = new ABA_STRING (master_);
      master_→assignParameter(*(hostname_[i]), help);
      if (myHostname_ ≡ hostname(i)) {
        if (hostId_ ≥ 0) {
          master_→err() ≪ "ABA_PARMASTER::initializeParameters(): host " ≪ myHostname_ ≪
              " is included in .abacus more than once!" ≪ endl;
          exit(Fatal);
        }
        hostId_ = i;
      }
    }
    if (hostId_ ≡ -1) {
      master_→err() ≪ "ABA_PARMASTER::initializeParameters(): host " ≪ myHostname_ ≪
          " is missing in .abacus!" ≪ endl;
      exit(Fatal);
    }
    master_→assignParameter(notifyPort_, "ParallelNotifyPort", 1, INT_MAX);
    master_→assignParameter(subserverPort_, "ParallelSubproblemPort", 1, INT_MAX);
    master_→assignParameter(balancerPort_, "ParallelBalancerPort", 1, INT_MAX);
    master_→assignParameter(bestFirstTolerance_, "ParallelBestFirstTolerance", 0, 100, 0);
    hostDualBounds_.initialize(hostCount_, master_→optSense()→min());
    openSubBest_.initialize(hostCount_, master_→optSense()→min());
    static int once = 0;
    if (once) return;
    once = 1;      /* run the following stuff only once */
      /* create new notification server with hostCount_ threads */
    notify_ = new ABA_NOTIFYSERVER (master_, notifyPort_, hostCount_);
    notify_→open(0);      /* create notification streams and connect with server */
    notifyStreams_ = new ACE_SOCK_Stream [hostCount_];
    for (int i = 0; i < hostCount_; i++) {
      char *name = hostname(i).string();
      ACE_INET_Addr addr(notifyPort_, name);
      ACE_SOCK_Connector connector;
      while (connector.connect(notifyStreams_[i], addr) ≡ -1) {
        if (debug(DEBUG_SOCKET)) master_→out() ≪ "DEBUG_SOCKET: server " ≪ name ≪ ":" ≪
            notifyPort_ ≪ " not available, retrying..." ≪ endl;
        sleep(1);
```

```
    }
  }
  if (debug(DEBUG_SOCKET)) master_→out() ≪ "DEBUG_SOCKET: notification " ≪
      "connections successfully established." ≪ endl;    /* create subproblem server */
  subserver_ = new ABA_SUBSERVER (master_, subserverPort_, 1);
  subserver_→open(0);    /* create balancing server */
  balancer_ = new ABA_BALANCER (master_, balancerPort_, 1);
  balancer_→open(0);
}
```

6. The function *setDefaultParameters*( ) sets default values of the parameters specific to the parallel version of ABACUS.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::setDefaultParameters()
  {
    master_→insertParameter("ParallelDebugLevel", "0");
  }
```

7. The function *printParameters*( ) prints the settings of the parameters specific to the parallel version of ABACUS.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::printParameters()
  {
    master_→out() ≪ endl ≪ "Parallel Parameters:" ≪ endl ≪ endl;
    master_→out() ≪ "  Debug Level                              : ";
    master_→out() ≪ debugLevel_ ≪ endl;
    master_→out() ≪ "  Connect Timeout                          : ";
    master_→out() ≪ connectTimeout_ ≪ endl;
    master_→out() ≪ "  Number of parallel hosts                 : ";
    master_→out() ≪ hostCount_ ≪ endl;
    for (int i = 0; i < hostCount_; i++) {
      master_→out() ≪ "  Hostname " ≪ setWidth(2) ≪ i ≪ "                              : " ≪
          hostname(i) ≪ endl;
    }
    master_→out() ≪ "  Notify Port                              : ";
    master_→out() ≪ notifyPort_ ≪ endl;
    master_→out() ≪ "  Subproblem Server Port                   : ";
    master_→out() ≪ subserverPort_ ≪ endl;
    master_→out() ≪ "  Balancer Port                            : ";
    master_→out() ≪ balancerPort_ ≪ endl;
    master_→out() ≪ "  Best First Search Tolerance              : ";
    master_→out() ≪ bestFirstTolerance_ ≪ endl;
  }
```

8. The function *outputStatistics*( ) prints statistics specific to the parallel version of ABACUS.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::outputStatistics( )
  {
    const int w = 6;
    master_→out( ) ≪ endl;
    master_→out( ) ≪ "Parallel␣Statistices" ≪ endl ≪ endl;
    master_→out( ) ≪ "␣␣Number␣of␣Subproblems␣sent␣␣␣␣␣␣␣␣:␣";
    master_→out( ) ≪ setWidth(w) ≪ subSentCount_ ≪ endl;
    master_→out( ) ≪ "␣␣Number␣of␣Subproblems␣received␣␣␣␣:␣";
    master_→out( ) ≪ setWidth(w) ≪ subReceivedCount_ ≪ endl;
    master_→out( ) ≪ "␣␣idle␣time␣before␣first␣subproblem␣:␣";
    master_→out( ) ≪ setWidth(w) ≪ idleCowTimeFirst_ ≪ endl;
    master_→out( ) ≪ "␣␣idle␣time␣during␣optimization␣␣␣␣␣:␣";
    master_→out( ) ≪ setWidth(w) ≪ idleCowTimeMiddle_ ≪ endl;
    master_→out( ) ≪ "␣␣idle␣time␣after␣last␣subproblem␣␣␣:␣";
    master_→out( ) ≪ setWidth(w) ≪ idleCowTimeLast_ ≪ endl;
  }
```

9. The function *registerPool*( ) registers an **ABA_POOL** with the master.

Arguments:

pool

A pointer to the **ABA_POOL**.

Return Value:

An integer identifying the pool.

```
⟨ parmaster.cc 2 ⟩ +≡
  int ABA_PARMASTER::registerPool(void *pool)
  {
    int n = registeredPools_.size( );
    int i = lastRegisteredPool_;
    do {
      i++;
      if (i ≡ n) i = 0;
      if (¬registeredPools_[i]) {
        registeredPools_[i] = pool;
        lastRegisteredPool_ = i;
        return i;
      }
    } while (i ≠ lastRegisteredPool_);
    registeredPools_.realloc(2 * n);
    registeredPools_[n] = pool;
    registeredPools_.set(n + 1, 2 * n − 1, 0);
    lastRegisteredPool_ = n;
    return n;
  }
```

10. The function *unregisterPool*( ) unregisters a previously registered **ABA_POOL**.

Arguments:

index

The index of the **ABA_POOL** which was returned by the function *registerPool*( ) when the **ABA_POOL** was registered.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::unregisterPool(int index)
  {
    registeredPools_[index] = 0;
  }
```

11. The function *getPool*( ).

Return Value:

A Pointer to the registered **ABA_POOL**.

Arguments:

index

The index of the registered **ABA_POOL**.

```
⟨ parmaster.cc 2 ⟩ +≡
  void *ABA_PARMASTER::getPool(int index)
  {
    return registeredPools_[index];
  }
```

12. The function *connectService*( ).

Arguments:

port

The port number of the service to connect to.

msg

The **ABA_MESSAGE** where the connection will be established into.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::connectService(int port, ABA_MESSAGE &msg, int destId)
  {
    char *name = hostname(destId).string();
    ACE_INET_Addr addr(port, name);
    ACE_SOCK_Connector connector;
    while (connector.connect(msg.stream(), addr) ≡ −1) {
      if (debug(DEBUG_SOCKET)) master_→out() ≪ "DEBUG_SOCKET: server " ≪ name ≪ ":" ≪ port ≪
          " not available, retrying..." ≪ endl;
      sleep(1);
    }
  }
```

13. The function *newHostDualBound*( ) updates the dual bound of this host, i.e. the minimum (maximum) dual bound of all subproblems of this host. The new bound is broadcast to all other hosts if it has changed. The global dual bound stored in the master is also updated, if needed.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::newHostDualBound(double x)
  {
    newHostDualBoundMutex_.acquire( );
    if (hostDualBounds_.better(hostId_, x)) {
      hostDualBounds_.insert(hostId_, x);
      double newDual = hostDualBounds_.best( );
      if (master_→betterDual(newDual)) master_→dualBound(newDual);
      if (debug(DEBUG_NOTIFICATION)) master_→out( ) ≪ "DEBUG_NOTIFICATION:␣new␣dual␣bound␣" ≪
          x ≪ "␣broadcast␣by␣" ≪ hostname( ) ≪ endl;
      ABA_BROADCAST bcast(master_);
      bcast.pack(ABA_NOTIFYSERVER::NewHostDualBoundTag);
      bcast.pack(hostId_);
      bcast.pack(x);
      bcast.send( );
    }
    newHostDualBoundMutex_.release( );
  }
```

14. This version of the function *newHostDualBound*( ) receives a new dual bound of another host from a message and updates the corresponding element in the array *hostDualBounds_*. The global dual bound stored in the master is also updated, if needed.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::newHostDualBound(ABA_MESSAGE &msg)
  {
    int id;
    double x;
    msg.unpack(id);
    msg.unpack(x);
    if (debug(DEBUG_NOTIFICATION)) master_→out( ) ≪ "DEBUG_NOTIFICATION:␣new␣dual␣bound␣" ≪
        x ≪ "␣received␣from␣" ≪ hostname(id) ≪ endl;
    newHostDualBoundMutex_.acquire( );
    if (hostDualBounds_.better(id, x)) {
      hostDualBounds_.insert(id, x);
      double newDual = hostDualBounds_.best( );
      if (master_→betterDual(newDual)) master_→dualBound(newDual);
    }
    newHostDualBoundMutex_.release( );
  }
```

15. The function *newPrimalBound*( ) broadcasts a new primal bound to all hosts.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::newPrimalBound(double x)
  {
    if (debug(DEBUG_NOTIFICATION))
      master_→out( ) ≪ "new␣primal␣bound␣" ≪ x ≪ "␣broadcast␣by␣" ≪ hostname( ) ≪ endl;
    ABA_BROADCAST bcast(master_);
    bcast.pack(ABA_NOTIFYSERVER::NewPrimalBoundTag);
    bcast.pack(x);
    bcast.send( );
  }
```

16. This version of the function *newPrimalBound*( ) receives a new primal bound.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::newPrimalBound(ABA_MESSAGE &msg)
  {
    double x;
    msg.unpack(x);
    if (debug(DEBUG_NOTIFICATION))
      master_→out( ) ≪ "DEBUG_NOTIFICATION:␣new␣primal␣bound␣" ≪ x ≪ "␣received." ≪ endl;
    newPrimalBoundMutex_.acquire( );
    if (master_→betterPrimal(x)) master_→primalBound(x, false);
    newPrimalBoundMutex_.release( );
  }
```

17. The function *newOpenSubCount*( ) broadcasts the new number of open subproblems and the best bound of all subproblems in the openSub list of this host to all other hosts if it has changed.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::newOpenSubCount(int n, double best)
  {
    newOpenSubCountMutex_.acquire( );
    if (n ≠ openSubCount_[hostId_]) {
      openSubCount_[hostId_] = n;
      openSubBest_.insert(hostId_, best);      /* trigger load balancing */
      master_→openSub( )→triggerSelect( );
      if (debug(DEBUG_NOTIFICATION)) master_→out( ) ≪ "DEBUG_NOTIFICATION:␣" ≪ hostname( ) ≪
            "␣broadcasts:␣#subproblems=" ≪ n ≪ ",␣best=" ≪ best ≪ endl;
      ABA_BROADCAST bcast(master_);
      bcast.pack(ABA_NOTIFYSERVER::NewOpenSubCountTag);
      bcast.pack(hostId_);
      bcast.pack(n);
      bcast.pack(best);
      bcast.send( );
    }
    newOpenSubCountMutex_.release( );
  }
```

18. This version of the function *newOpenSubCount*( ) receives the new number of open subproblems and the best bound of the openSub list broadcast by another host.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::newOpenSubCount(ABA_MESSAGE &msg)
  {
    int id, n;
    double best;
    msg.unpack(id);
    msg.unpack(n);
    msg.unpack(best);
    if (debug(DEBUG_NOTIFICATION)) master_->out( ) << "DEBUG_NOTIFICATION: received from "
        << hostname(id) << ": #subproblems=" << n << ", best=" << best << endl;
    newOpenSubCountMutex_.acquire( );
    openSubCount_[id] = n;
    openSubBest_.insert(id, best);
    newOpenSubCountMutex_.release( );      /* trigger load balancing */
    master_->openSub( )->triggerSelect( );
  }
```

19. The function *balance*( ) is called before a new subproblem is selected for optimization to perform a loadbalancing step. *bestFirstTolerance_* defines the allowed deviation of the dual bound of the best subproblem at this host compared to the global best dual bound. If the best locally available subproblem is not within the tolerance, the best subproblem available at some other host is requested for. The location of the best subproblem is determined by inspecting *hostDualBounds_*.

Return Value:
true
if successful
false
otherwise.

```
⟨ parmaster.cc 2 ⟩ +≡
  bool ABA_PARMASTER::balance( )
  {
    int host;
    double best = openSubBest_.best(&host);
    if (host < 0) return false;
    if (host ≡ hostId_)      /* best subproblem is at this host */
      return true;
    if (¬master_→openSub( )→empty( )) {
      /* check whether the locally avalibale subproblem is good enough */
      double total = master_→primalBound( ) − best;
      double dist = master_→openSub( )→dualBound( ) − best;
      if (master_→optSense( )→max( )) {
        total = −total;
        dist = −dist;
      }
      if (debug(DEBUG_BALANCER))
        master_→out( ) ≪ "DEBUG_BALANCER: globalBest=" ≪ best ≪ ", primal=" ≪
            master_→primalBound( ) ≪ ", localBest=" ≪ master_→openSub( )→dualBound( ) ≪ endl;
      if (total ≤ 0) return false;
      if (dist ∗ 100 ≤ total ∗ bestFirstTolerance_) return true;
    }
    if (debug(DEBUG_BALANCER))
      master_→out( ) ≪ "DEBUG_BALANCER: requesting subproblem with dual bound " ≪ best ≪
          " from " ≪ hostname(host) ≪ endl;
    ABA_MESSAGE msg;
    connectService(balancerPort_, msg, host);
    msg.pack(hostId_);      /* the requesting host */
    msg.pack(best);      /* the requested dual bound */
    msg.send( );
    bool success;
    msg.receive( );
    msg.unpack(success);
    if (debug(DEBUG_BALANCER)) master_→out( ) ≪ "DEBUG_BALANCER: " ≪ hostname(host) ≪ (success ?
        " confirms to send " : " doesn't send ") ≪ "a subproblem." ≪ endl;
    if (success) {
      ABA_SUB ∗sub = master_→unpackSub(msg);
      if (debug(DEBUG_BALANCER)) {
        master_→out( ) ≪ "DEBUG_BALANCER: subproblem ";
        printId(sub→id( ));
```

```
        master_→out( ) ≪ "␣received␣from␣" ≪ hostname(host) ≪ endl;
      }
      master_→openSub( )→insert(sub);
    }
    return success;
  }
```

20. The function *terminationCheck* tests, if all hosts are idle and no subproblems are staying arround anywhere. This is done by a two phase protocol initiated by host 0. In the first phase host 0 broadcasts a state request to all other hosts. Each host sets its local flag *terminationOk_* if it is idle and want's to terminate. This local flag is cleared immediately if work (e.g. a subproblem) is received since that time. Then each host sends a reply with dummy data. After all hosts replies are received the second phase is started. Host 0 broadcasts a state request again. This time, each host replies the value of its local flag *terminationOk_*. The flag is true if no work (i.e. subproblem) was received since the first phase. If all hosts reply true the system is out of work and termination can be initiated.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::terminationCheck()
  {
    assert(hostId_ ≡ 0);
    if (debug(DEBUG_TERMINATION))
      master_→out() ≪ "DEBUG_TERMINATION:␣terminationCheck()␣called" ≪ endl;
    for (int i = 0; i < hostCount_; i++)
      if (openSubCount_[i] ≠ 0) return;      /* do not test for termination */      /* if the termination
          check is already running we set a flag to */      /* remember that we have to check again. */
    if (terminationCheckMutex_.tryacquire() ≡ −1) {
      terminationCheckAgain_ = true;
      return;
    }
    if (hasTerminated_) {
      terminationCheckMutex_.release();
      return;
    }
    do {
      terminationCheckAgain_ = false;
      if (debug(DEBUG_TERMINATION))
        master_→out() ≪ "DEBUG_TERMINATION:␣Phase␣I␣started..." ≪ endl;
      int idle;      /* terminationOk_ is set, if the processor is idle */
      startTerminationCheck();
      {
        ABA_BROADCAST bcast(master_);
        bcast.pack(ABA_NOTIFYSERVER::TerminationCheckTag);
        bcast.send();
        bcast.receiveReply();      /* receive accumulated dummy reply */
        if (debug(DEBUG_TERMINATION))
          master_→out() ≪ "DEBUG_TERMINATION:␣Phase␣II␣started..." ≪ endl;
        bcast.pack(0);      /* send some dummy data */
        bcast.send();
        idle = bcast.receiveReply();      /* receive accumulated reply */
      }
      if (idle < hostCount_ − 1) {
        terminationMutex_.acquire();
        terminationOk_ = false;
        terminationMutex_.release();
      }
      if (debug(DEBUG_TERMINATION))
        master_→out() ≪ "DEBUG_TERMINATION:␣terminationOk␣=␣" ≪ terminationOk_ ≪ endl;
    } while (terminationCheckAgain_ ∧ ¬terminationOk_);
    if (terminationOk_) hasTerminated_ = true;
```

```
    terminationCheckMutex_.release();      /* should the system be terminated? */
    if (terminationOk_) {
      terminate();
      ABA_BROADCAST bcast(master_);
      bcast.pack(ABA_NOTIFYSERVER::TerminateTag);
      bcast.send();
    }
  }
```

21. This version of the function *terminationCheck*( ) handles the termination test phases I and II.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::terminationCheck(ABA_MESSAGE &msg)
  {
    if (debug(DEBUG_TERMINATION))
      master_→out() ≪ "DEBUG_TERMINATION: handling phase I." ≪ endl;
        /* terminationOk_ is set, if the processor is idle */
    startTerminationCheck();      /* send dummy reply for synchronisation */
    msg.pack(false);
    msg.send();      /* terminateFlag_ is reset asynchronously when a subproblem is received */
      /* wait for phase II */
    msg.receive();
    int dummy;
    msg.unpack(dummy);
    if (debug(DEBUG_TERMINATION)) master_→out() ≪ "DEBUG_TERMINATION: handling phase II, " ≪
        "terminationOk = " ≪ terminationOk_ ≪ endl;      /* send reply */
    msg.pack(terminationOk_);
    msg.send();
  }
```

22. The function *startTerminationCheck*( ) sets the local flag *terminationOk_* if this host is idle.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::startTerminationCheck()
  {
    terminationMutex_.acquire();
    if (workCount_ ≡ 0) terminationOk_ = true;
    terminationMutex_.release();
  }
```

23. The function *incWorkCount*( ) has to be called if the host got new work, e.g. if a new subproblem is created. The local variable *workCount_* is incremented and the flag *terminationOk_* is cleared.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::incWorkCount()
  {
    terminationMutex_.acquire();
    workCount_++;
    terminationOk_ = false;
    terminationMutex_.release();
  }
```

24. The function *decWorkCount*( ) has to be called if the host has finished some work, e.g. if a new subproblem is created. The local variable *workCount_* is decremented. If *workCount_* is zero the processor is idle.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::decWorkCount( )
  {
    terminationMutex_.acquire( );
    workCount_--;
    terminationMutex_.release( );
    if (workCount_ ≡ 0) {
      ABA_NOTIFICATION msg(master_, 0);
      msg.pack(ABA_NOTIFYSERVER::TriggerTerminationCheckTag);
      msg.send( );
    }
  }
```

25. The function *terminate*( ) terminates all threads.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::terminate( )
  {
    if (debug(DEBUG_TERMINATION))
      master_→out( ) ≪ "DEBUG_TERMINATION: Terminating all Threads..." ≪ endl;
    master_→openSub( )→terminate( );
    if (debug(DEBUG_TERMINATION)) master_→out( ) ≪ "DEBUG_TERMINATION: finished." ≪ endl;
  }
```

26. The function *newId*( ) requests a new ID for the subproblem. If *master_→vbcLog*( ) is true a globally unique ID is requested from host 0 to get a continuous sequence of IDs. Otherwise the ID is generated by adding 1000000 ∗ *hostId_* a local counter.

```
⟨ parmaster.cc 2 ⟩ +≡
  int ABA_PARMASTER::newId(int fatherId)
  {
    if (master_→vbcLog( ) ≠ ABA_MASTER::NoVbc) {
      if (hostId_ ≡ 0) {
        idCounterMutex_.acquire( );
        int id = idCounter_++;
        master_→treeInterfaceNewNode(fatherId, id);
        idCounterMutex_.release( );
        return id;
      }     /* request unique Id from host 0 */
      ABA_NOTIFICATION msg(master_, 0);
      msg.pack(ABA_NOTIFYSERVER::NewIdTag);
      msg.pack(fatherId);
      msg.send( );
      msg.receive( );
      return msg.unpackInt( );
    }
    else {
      idCounterMutex_.acquire( );
      int id = idCounter_++;
      idCounterMutex_.release( );
      return hostId_ ∗ 1000000 + id;
    }
  }
```

27. The function *printId*( ) prints an ID. If the IDs are not requested continuous (vbcLog!=NoVbc) a pair consisting of the nmae of the host which has generated the subproblem and the local number of the subproblem is output.

```
⟨ parmaster.cc 2 ⟩ +≡
  void ABA_PARMASTER::printId(int id)
  {
    if (master_→vbcLog( ) ≡ ABA_MASTER::NoVbc)
      master_→out( ) ≪ "(" ≪ hostname(id/1000000) ≪ "," ≪ id % 1000000 ≪ ")";
    else master_→out( ) ≪ id;
  }
```

28. MULTITHREADED SERVER.

The class **ABA_MTSERVER** is an abstract base class of a multithreaded server, which listens on a port for messages of type **ABA_MESSAGE**. The pure virtual function *svcMessage* must be defined in a subclass.

```
⟨ mtserver.h 28 ⟩ ≡
#ifndef ABA_MTSERVER_H
#define ABA_MTSERVER_H
#include "abacus/abacusroot.h"
#include "abacus/message.h"
#include <ace/Task.h>
#include <ace/SOCK_Acceptor.h>
  class ABA_MASTER;
  class ABA_MTSERVER : public ACE_Task⟨ACE_MT_SYNCH⟩, public ABA_ABACUSROOT {
  public:
    ABA_MTSERVER(ABA_MASTER *master, int port, int nThreads);
    ~ABA_MTSERVER();
    int open(void *);
    int close(u_long);
    int svc();
  protected:
    ABA_MASTER *master_;
    int port_;
    int nThreads_;
    ACE_SOCK_Acceptor acceptor_;
    virtual bool svcMessage(ABA_MESSAGE &msg) = 0;
  };
  inline int ABA_MTSERVER::close(u_long)
  {
    return 0;
  }
#endif    /* ¬ABA_MTSERVER_H */
```

29. The member functions are defined in the file `mtserver.cc`.

```
⟨ mtserver.cc 29 ⟩ ≡
#include "abacus/mtserver.h"
#include "abacus/parmaster.h"
#include "abacus/master.h"
#include <ace/SOCK_Stream.h>
#include <ace/INET_Addr.h>
```

See also sections 30, 31, 32, and 33.

30. The constructor.

Arguments:

master
A pointer to the master object.

port
The port number on which the server is listening.

nThreads
The number of threads.

```
⟨ mtserver.cc 29 ⟩ +≡
ABA_MTSERVER::ABA_MTSERVER(ABA_MASTER *master, int port, int nThreads):
  master_(master),
  port_(port),
  nThreads_(nThreads)
  { }
```

31. The destructor.

```
⟨ mtserver.cc 29 ⟩ +≡
  ABA_MTSERVER::~ABA_MTSERVER()
  { }
```

32. The definition of the *open* function of **ACE_Task**. This function begins listening for connections on *port_*. *nThreads_* new threads are started which all will all accept connections on this port.

```
⟨ mtserver.cc 29 ⟩ +≡
  int ABA_MTSERVER::open(void *)
  {
    ACE_INET_Addr addr(port_);
    if (acceptor_.open(addr, 1) ≡ −1) {
      master_→err() ≪ "ABA_MTSERVER::open():␣listening␣on␣port␣" ≪ port_ ≪ "␣failed!" ≪ endl;
      exit(Fatal);
    }
    activate(THR_NEW_LWP, nThreads_);
    return 0;
  }
```

33. The function *svc* is the entry point of all threads.

```
⟨ mtserver.cc 29 ⟩ +≡
  int ABA_MTSERVER::svc()
  {
    ACE_Time_Value timeout(1);
    while (ACE_Thread_Manager::instance()→testcancel(ACE_Thread::self()) ≡ 0) {
      ABA_MESSAGE msg;
      ACE_INET_Addr addr;
      int ret = acceptor_.accept(msg.stream(), &addr, &timeout);
      if (ret ≡ 0)
        if (¬svcMessage(msg)) break;
    }
    return 0;
  }
```

34. The pure virtual function *svcMessage* has to be defined in a subclass. This function must receive the message and service the connection. When this function returns the connection will be closed.

bool ABA_MTSERVER::svcMessage(ABA_MESSAGE**m**sg)

35. NOTIFICATION SERVER.

The class **ABA_NOTIFYSERVER** is a server for receiving notification messages. If a notification message of this class is called, messages representing the particular information are sent to all other hosts of the system. The *scvMessage* method of this class runs in a separate thread and listens for such messages. If a notification message is received, the related state information is updated.

This class is used to efficently broadcast information which should be known globally, for example new dual bounds.

```
⟨ notifyserver.h 35 ⟩ ≡
#ifndef ABA_NOTIFYSERVER_H
#define ABA_NOTIFYSERVER_H
#include "abacus/mtserver.h"
  class ABA_NOTIFYSERVER : public ABA_MTSERVER {
  public:
    enum NOTIFYTAG {
      InvalidTag, NewHostDualBoundTag, NewPrimalBoundTag, NewOpenSubCountTag,
          TerminationCheckTag, TriggerTerminationCheckTag, TerminateTag, NewIdTag,
          TreeInterfaceTag, ShutdownTag
    };
    ABA_NOTIFYSERVER(ABA_MASTER *master, int port, int nThreads);
  protected:
    virtual bool svcMessage(ABA_MESSAGE &msg);
  };
inline ABA_NOTIFYSERVER::ABA_NOTIFYSERVER(ABA_MASTER *master, int port, int
        nThreads): ABA_MTSERVER(master, port, nThreads)
  {}
#endif    /* ¬ABA_NOTIFYSERVER_H */
```

36. The member functions are defined in the file `notifyserver.cc`.

```
⟨ notifyserver.cc 36 ⟩ ≡
#include "abacus/notifyserver.h"
#include "abacus/master.h"
#include "abacus/parmaster.h"
#include "abacus/broadcast.h"
```

See also section 37.

37. The function *svcMessage* receives a notification and handles it. For each peer exists a separate thread running this function.

```
⟨ notifyserver.cc 36 ⟩ +≡
  bool ABA_NOTIFYSERVER::svcMessage(ABA_MESSAGE &msg)
  {
    while (1) {
      msg.receive();
      int type = msg.unpackInt();
      switch (type) {
      case NewHostDualBoundTag: master_→parmaster()→newHostDualBound(msg);
        break;
      case NewPrimalBoundTag: master_→parmaster()→newPrimalBound(msg);
        break;
      case NewOpenSubCountTag: master_→parmaster()→newOpenSubCount(msg);
        break;
      case TerminationCheckTag: master_→parmaster()→terminationCheck(msg);
        break;
      case TriggerTerminationCheckTag: assert(master_→parmaster()→hostId() ≡ 0);
        master_→parmaster()→terminationCheck();
        break;
      case TerminateTag: master_→parmaster()→terminate();
        break;
      case ShutdownTag:
        {
          static ACE_Thread_Mutex m;
          m.acquire();
          if (debug(DEBUG_TERMINATION))
            master_→out() ≪ "DEBUG_TERMINATION:␣shutdown␣notification␣thread.\n";
          m.release();
          msg.pack(false);
          msg.send();      /* send reply */
          return false;
        }
      case NewIdTag:
        {
          assert(master_→parmaster()→hostId() ≡ 0);
          int fatherId = msg.unpackInt();      /* call newId() at host 0 */
          int id = master_→parmaster()→newId(fatherId);      /* a reply is sent containing the result */
          msg.pack(id);
          msg.send();
          break;
        }
      case TreeInterfaceTag:
        {
          assert(master_→parmaster()→hostId() ≡ 0);
          char info[256];
          int len = msg.unpackInt();
          assert(len < 256);
          msg.unpack(info, len);
          info[len] = 0;
```

```
            master_→writeTreeInterface(info);
            break;
          }
        }
      }
    }
    return true;
  }
```

38. SUBPROBLEM SERVER.

The class **ABA_SUBSERVER** is a multithreaded server for solving subproblems.

```
⟨ subserver.h 38 ⟩ ≡
#ifndef ABA_SUBSERVER_H
#define ABA_SUBSERVER_H
#include "abacus/mtserver.h"
  class ABA_SUBSERVER : public ABA_MTSERVER {
  public:
    ABA_SUBSERVER(ABA_MASTER *master, int port, int nThreads);
  protected:
    virtual bool svcMessage(ABA_MESSAGE &msg);
  };
inline ABA_SUBSERVER::ABA_SUBSERVER(ABA_MASTER *master, int port, int nThreads):
      ABA_MTSERVER(master, port, nThreads)
  { }
#endif     /* ¬ABA_SUBSERVER_H */
```

39. The member functions are defined in the file `subserver.cc`.

```
⟨ subserver.cc 39 ⟩ ≡
#include "abacus/subserver.h"
#include "abacus/master.h"
#include "abacus/parmaster.h"
#include "abacus/sub.h"
#include "abacus/opensub.h"
#include "abacus/debug.h"
```

See also section 40.

40. The function *svcMessage* receives a subproblem and solves it.

```
⟨ subserver.cc 39 ⟩ +≡
  bool ABA_SUBSERVER::svcMessage(ABA_MESSAGE &msg)
  {
    if (debug(DEBUG_SUBSERVER))
      master_→out() ≪ "DEBUG_SUBSERVER: receiving a subproblem..." ≪ endl;
    msg.receive();
    ABA_SUB *sub = master_→unpackSub(msg);
    master_→openSub()→insert(sub);
    if (debug(DEBUG_SUBSERVER)) {
      master_→out() ≪ "DEBUG_SUBSERVER: subproblem ";
      master_→parmaster()→printId(sub→id());
      master_→out() ≪ " received and inserted into openSub list." ≪ endl;
    }
    return true;
  }
```

41. LOAD BALANCING THREAD.

The class **ABA_BALANCER** is used for load balancing. Some host can send a request for a subproblem of specified dual bound. If such a subproblem is contained in the list of open subproblems of this host, the balancer will send that subproblem to the requester.

⟨ balancer.h 41 ⟩ ≡

```
#ifndef ABA_BALANCER_H
#define ABA_BALANCER_H
#include "abacus/mtserver.h"
  class ABA_BALANCER : public ABA_MTSERVER {
  public:
    ABA_BALANCER(ABA_MASTER *master, int port, int nThreads);
  protected:
    virtual bool svcMessage(ABA_MESSAGE &msg);
  };
inline ABA_BALANCER::ABA_BALANCER(ABA_MASTER *master, int port, int nThreads):
      ABA_MTSERVER(master, port, nThreads)
  { }
#endif     /* ¬ABA_BALANCER_H */
```

42. The member functions are defined in the file `balancer.cc`.

⟨ balancer.cc 42 ⟩ ≡

```
#include "abacus/balancer.h"
#include "abacus/master.h"
#include "abacus/parmaster.h"
#include "abacus/sub.h"
#include "abacus/opensub.h"
#include "abacus/debug.h"
```

See also section 43.

43. The function *svcMessage* receives requests for subproblems of some desired dual bound. If the host hold such a subproblem in its list of open subproblems, the subproblem is sent to the requesting host.

```
⟨ balancer.cc 42 ⟩ +≡
  bool ABA_BALANCER::svcMessage(ABA_MESSAGE &msg)
  {
    msg.receive();
    int host;
    double requestedDualBound;
    msg.unpack(host);
    msg.unpack(requestedDualBound);
    if (debug(DEBUG_BALANCER))
      master_->out() << "DEBUG_BALANCER: " << master_->parmaster()->hostname(host) <<
          " requests a subproblem with dual bound " << requestedDualBound << endl;
    assert(host != master_->parmaster()->hostId());
    ABA_SUB *s = master_->openSub()->getSubproblemWithBound(requestedDualBound);
    if (s) {
      if (debug(DEBUG_BALANCER)) {
        master_->out() << "DEBUG_BALANCER: sending subproblem ";
        master_->parmaster()->printId(s->id());
        master_->out() << " to " << master_->parmaster()->hostname(host) << " ..." << endl;
      }
      msg.pack(true);
      s->pack(msg);
      msg.send();
      delete s;
    }
    else {
      if (debug(DEBUG_BALANCER))
        master_->out() << "DEBUG_BALANCER: no such problem available!" << endl;
      msg.pack(false);
      msg.send();
    }
    return true;
  }
```

44. MESSAGE BASE CLASS.

The **ABA_MESSAGEBASE** class implements a message buffer for sending arbitrary objects over a communication stream. It fully encapsulates the underlaying XDR stream which provides machine independant data representation. The information needed to recreate an object has to be packed in the message buffer by *pack*( ) functions. Afterwards the message has to be sent by *send*( ). The receiver calls *receive*( ) and then *unpack*( ) to obtain the data. We have chosen message buffers to provide a mechanism for sending objects of derived subclasses and sending data of variable size. Large objects are sent in message chunks. The user has to provide the member functions *pack*( ) and *unpack*( ) in derived classes as well as a constructor which constructs an object of that class from a message buffer.

⟨ messagebase.h 44 ⟩ ≡

```
#ifndef ABA_MESSAGEBASE_H
#define ABA_MESSAGEBASE_H

#include <assert.h>
#include <rpc/rpc.h>

  class ABA_MESSAGEBASE
  {
  public:
    ABA_MESSAGEBASE(int (*readit)(void *, char *, int), int (*writeit)(void *, char *, int));
    ~ABA_MESSAGEBASE( );

    void send( );
    void receive( );

    void pack(bool a);
    void pack(const bool a[], int count);
    void pack(char a);
    void pack(const char a[], int count);
    void pack(short a);
    void pack(const short a[], int count);
    void pack(int a);
    void pack(const int a[], int count);
    void pack(long a);
    void pack(const long a[], int count);
    void pack(unsigned char a);
    void pack(const unsigned char a[], int count);
    void pack(unsigned short a);
    void pack(const unsigned short a[], int count);
    void pack(unsigned int a);
    void pack(const unsigned int a[], int count);
    void pack(unsigned long a);
    void pack(const unsigned long a[], int count);
    void pack(float a);
    void pack(const float a[], int count);
    void pack(double a);
    void pack(const double a[], int count);      /* ... */

    void unpack(bool &a);
    void unpack(bool a[], int count);
    void unpack(char &a);
    void unpack(char a[], int count);
    void unpack(short &a);
    void unpack(short a[], int count);
    void unpack(int &a);
    int unpackInt( );
```

```
    void unpack(int a[], int count);
    void unpack(long &a);
    void unpack(long a[], int count);
    void unpack(unsigned char &a);
    void unpack(unsigned char a[], int count);
    void unpack(unsigned short &a);
    void unpack(unsigned short a[], int count);
    void unpack(unsigned int &a);
    void unpack(unsigned int a[], int count);
    void unpack(unsigned long &a);
    void unpack(unsigned long a[], int count);
    void unpack(float &a);
    void unpack(float a[], int count);
    void unpack(double &a);
    void unpack(double a[], int count);      /* ... */
  protected:
    XDR xdr_;
}
;
```

See also sections 45 and 46.

45. We provide *pack*( ) functions for some basic datatypes. Each user defined class whose objects will be transferred between processors has to provide an implementation of a *pack*( ) function.

⟨ messagebase.h 44 ⟩ +≡

```
inline void ABA_MESSAGEBASE::pack(bool a)
{
  xdr_.x_op = XDR_ENCODE;
  assert(sizeof(bool) ≡ sizeof (bool_t));
  bool_t b = a;
  int ret = xdr_bool(&xdr_, &b);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const bool a[], int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(bool), xdr_bool);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(char a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_char(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const char a[], int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_bytes(&xdr_, (char **) &a, (unsigned int *) &count, count);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(short a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_short(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const short a[], int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(short), xdr_short);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(int a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_int(&xdr_, &a);
  assert(ret);
}
```

```
inline void ABA_MESSAGEBASE::pack(const int a[],int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_array(&xdr_,(char **) &a,(unsigned int *) &count,count,sizeof(int),xdr_int);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(long a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_long(&xdr_,&a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const long a[],int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_array(&xdr_,(char **) &a,(unsigned int *) &count,count,sizeof(long),xdr_long);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(unsigned char a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_u_char(&xdr_,&a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const unsigned char a[],int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_bytes(&xdr_,(char **) &a,(unsigned int *) &count,count);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(unsigned short a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_u_short(&xdr_,&a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const unsigned short a[],int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_array(&xdr_,(char **) &a,(unsigned int *) &count,count,sizeof(unsigned
      short),xdr_u_short);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(unsigned int a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_u_int(&xdr_,&a);
```

```
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const unsigned int a[], int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(unsigned
      int), xdr_u_int);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(unsigned long a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_u_long(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const unsigned long a[], int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(unsigned
      long), xdr_u_long);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(float a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_float(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const float a[], int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(float), xdr_float);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(double a)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_double(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::pack(const double a[], int count)
{
  xdr_.x_op = XDR_ENCODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(double), xdr_double);
  assert(ret);
}    /* ... */
```

46. We provide *unpack*( ) functions for some basic datatypes. Each user defined class whose objects will be transferred between processors has to provide an appropriate *unpack*( ) function.

```
⟨ messagebase.h 44 ⟩ +≡
  inline void ABA_MESSAGEBASE::unpack(bool &a){ assert(sizeof(bool) ≡ sizeof (bool_t));
      xdr_.x_op = XDR_DECODE; int ret = xdr_bool (&xdr_, ( bool_t * ) &a ) ;
      assert(ret); } inline void ABA_MESSAGEBASE::unpack(bool a[], int count)
      {
        xdr_.x_op = XDR_DECODE;
        int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(bool), xdr_bool);
        assert(ret);
      }
      inline void ABA_MESSAGEBASE::unpack(char &a)
      {
        xdr_.x_op = XDR_DECODE;
        int ret = xdr_char(&xdr_, &a);
        assert(ret);
      }
      inline void ABA_MESSAGEBASE::unpack(char a[], int count)
      {
        xdr_.x_op = XDR_DECODE;
        int ret = xdr_bytes(&xdr_, (char **) &a, (unsigned int *) &count, count);
        assert(ret);
      }
      inline void ABA_MESSAGEBASE::unpack(short &a)
      {
        xdr_.x_op = XDR_DECODE;
        int ret = xdr_short(&xdr_, &a);
        assert(ret);
      }
      inline void ABA_MESSAGEBASE::unpack(short a[], int count)
      {
        xdr_.x_op = XDR_DECODE;
        int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(short),
            xdr_short);
        assert(ret);
      }
      inline void ABA_MESSAGEBASE::unpack(int &a)
      {
        xdr_.x_op = XDR_DECODE;
        int ret = xdr_int(&xdr_, &a);
        assert(ret);
      }
      inline int ABA_MESSAGEBASE::unpackInt( )
      {
        int a;
        xdr_.x_op = XDR_DECODE;
        int ret = xdr_int(&xdr_, &a);
```

```
  assert(ret);
  return a;
}
inline void ABA_MESSAGEBASE::unpack(int a[], int count)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(int), xdr_int);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(long &a)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_long(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(long a[], int count)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(long), xdr_long);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(unsigned char &a)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_u_char(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(unsigned char a[], int count)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_bytes(&xdr_, (char **) &a, (unsigned int *) &count, count);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(unsigned short &a)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_u_short(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(unsigned short a[], int count)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(unsigned
      short), xdr_u_short);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(unsigned int &a)
```

```
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_u_int(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(unsigned int a[], int count)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(unsigned
      int), xdr_u_int);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(unsigned long &a)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_u_long(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(unsigned long a[], int count)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(unsigned
      long), xdr_u_long);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(float &a)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_float(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(float a[], int count)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(float), xdr_float);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(double &a)
{
  xdr_.x_op = XDR_DECODE;
  int ret = xdr_double(&xdr_, &a);
  assert(ret);
}
inline void ABA_MESSAGEBASE::unpack(double a[], int count)
{
  xdr_.x_op = XDR_DECODE;
```

```
    int ret = xdr_array(&xdr_, (char **) &a, (unsigned int *) &count, count, sizeof(double),
        xdr_double);
    assert(ret);
  }     /* ... */
#endif  /* ¬ABA_MESSAGEBASE_H */
```

47. The member functions are implemented in the file `messagebase.cc`.

```
⟨ messagebase.cc 47 ⟩ ≡
#include "abacus/messagebase.h"
```

See also sections 48, 49, 50, and 51.

48. The constructor. A message buffer encapsulates a bidirectional communication stream.

```
⟨ messagebase.cc 47 ⟩ +≡
  ABA_MESSAGEBASE::ABA_MESSAGEBASE(int(*readit)(void *, char *, int),
      int(*writeit)(void *, char *, int))
  {
    xdrrec_create(&xdr_, 0, 0, (char *) this, readit, writeit);
  }
```

49. The destructor.

```
⟨ messagebase.cc 47 ⟩ +≡
  ABA_MESSAGEBASE::~ABA_MESSAGEBASE()
  {
    xdr_destroy(&xdr_);
  }
```

50. The function *send*() sends the message buffer to the destination.

```
⟨ messagebase.cc 47 ⟩ +≡
  void ABA_MESSAGEBASE::send()
  {
    xdrrec_endofrecord(&xdr_, true);      /* flush */
  }
```

51. The function *receive*() receives the message buffer from the stream.

```
⟨ messagebase.cc 47 ⟩ +≡
  void ABA_MESSAGEBASE::receive()
  {
    xdrrec_skiprecord(&xdr_);      /* position stream */
  }
```

52. POINT TO POINT MESSAGE.

The **ABA_MESSAGE** class implements a message buffer for sending arbitrary objects over a communication stream. It fully encapsulates the underlaying XDR stream which provides machine independant data representation. The information needed to recreate an object has to be packed in the message buffer by *pack*( ) functions. Afterwards the message has to be sent by *send*( ). The receiver calls *receive*( ) and then *unpack*( ) to obtain the data. We have chosen message buffers to provide a mechanism for sending objects of derived subclasses and sending data of variable size. Large objects are sent in message chunks. The user has to provide the member functions *pack*( ) and *unpack*( ) in derived classes as well as a constructor which constructs an object of that class from a message buffer.

```
⟨ message.h 52 ⟩ ≡
#ifndef ABA_MESSAGE_H
#define ABA_MESSAGE_H
#include "abacus/messagebase.h"
#include <ace/SOCK_Stream.h>
  class ABA_MESSAGE : public ABA_MESSAGEBASE
  {
  public:
    ABA_MESSAGE( );
    ~ABA_MESSAGE( );
    ACE_SOCK_Stream &stream( );
  protected:
    static int readit(ABA_MESSAGE *msg, char *buf, int len);
    static int writeit(ABA_MESSAGE *msg, const char *buf, int len);
    ACE_SOCK_Stream stream_;
  }
  ;
  inline ACE_SOCK_Stream &ABA_MESSAGE::stream( )
  {
    return stream_;
  }
#endif     /* ¬ABA_MESSAGE_H */
```

53. The member functions are implemented in the file `message.cc`.

```
⟨ message.cc 53 ⟩ ≡
#include "abacus/message.h"
```

See also sections 54, 55, and 56.

54. Static helper functions to read/write data from/to the stream

```
⟨ message.cc 53 ⟩ +≡
  int ABA_MESSAGE::readit(ABA_MESSAGE *msg, char *buf, int len)
  {     /* receive and decode len1 (the length of the message) */
    XDRxdr;
    int len1;
    char buf_len1[4];
    int r = msg→stream_.recv_n(buf_len1, 4, 0);
    if (r ≠ 4) {
      ACE_ERROR((LM_ERROR, "%p\n", "Error in recv_n"));
      assert(0);
    }
    xdrmem_create(&xdr, buf_len1, 4, XDR_DECODE);
    xdr_int(&xdr, &len1);
    xdr_destroy(&xdr);
    assert(len1 ≤ len);      /* receive len1 bytes of data */
    r = msg→stream_.recv_n(buf, len1, 0);
    if (r ≠ len1) {
      ACE_ERROR((LM_ERROR, "%p\n", "Error in recv_n"));
      assert(0);
    }
    return r;
  }
  int ABA_MESSAGE::writeit(ABA_MESSAGE *msg, const char *buf, int len)
  {     /* encode and send the length */
    char buf_len[4];
    XDRxdr;
    xdrmem_create(&xdr, buf_len, 4, XDR_ENCODE);
    xdr_int(&xdr, &len);
    xdr_destroy(&xdr);
    int r = msg→stream_.send_n(buf_len, 4, 0);
    if (r ≠ 4) {
      ACE_ERROR((LM_ERROR, "%p\n", "Error in send_n"));
      assert(0);
    }      /* send the data */
    r = msg→stream_.send_n(buf, len, 0);
    if (r ≠ len) {
      ACE_ERROR((LM_ERROR, "%p\n", "Error in send_n"));
      assert(0);
    }
    return r;
  }
```

55. The constructor. A message buffer encapsulates a bidirectional communication stream. The underlaying ACE°_SOCK°_*Stream* has to be initialized later by using the member function *stream*().

```
⟨ message.cc 53 ⟩ +≡
ABA_MESSAGE::ABA_MESSAGE():
  ABA_MESSAGEBASE((int(*)(void *, char *, int))readit, (int(*)(void *, char *, int))writeit)
  {}
```

56. The destructor.

⟨ message.cc 53 ⟩ +≡

```
ABA_MESSAGE::~ABA_MESSAGE()
{
  stream_.close();
}
```

57. BROADCAST MESSAGE.

The **ABA_BROADCAST** class is derived from the **ABA_MESSAGE** class. It is used if one host wants to broadcast a message to all other hosts.

The notification streams set up by the parmaster are used for communication. The streams are not closed when an **ABA_BROADCAST** object is deleted. This also ensures that two broadcasts sent by the same processor one after the other are received in the same order as they were sent by every other processor.

The constructor of this class acquires a static mutex which is released in the destructor. This guarantees, that different broadcasts do not interfer each other.

The broadcast is currently implemented by sending a message to each other processor directly. This turned out to work well in case of a workstation cluster with ethernet bus topology. It could easily be changed to a message tree structure if congestion might occur at the sender.

A single boolean can be replied by each host. The number of hosts which replied true can be received and queried by the initiator of the bradcast by the function *receiveReply*( ). Using this feature a two phase termination detection mechanism is implemented.

```
⟨ broadcast.h 57 ⟩ ≡
#ifndef ABA_BROADCAST_H
#define ABA_BROADCAST_H
#include "abacus/messagebase.h"
#include <ace/Synch.h>
  class ABA_MASTER;
  class ABA_PARMASTER;
  class ABA_BROADCAST : public ABA_MESSAGEBASE {
  public:
    ABA_BROADCAST(const ABA_MASTER *master, bool includeMe = false);
    ~ABA_BROADCAST( );
    int receiveReply( );
  protected: ABA_BROADCAST(const ABA_MASTER *master, int(*readit)(void *, char
        *, int), int(*writeit)(void *, char *, int));
    static int broadcastit(ABA_BROADCAST *bcast, const char *buf, int len);
    static int broadcastreply(ABA_BROADCAST *bcast, char *buf, int len);
    static ACE_Thread_Mutex mutex_;
    ABA_PARMASTER *parmaster_;
    int reply_;
    bool includeMe_;
  };
#endif     /* ¬ABA_BROADCAST_H */
```

58. The member functions are implemented in the file `broadcast.cc`.

```
⟨ broadcast.cc 58 ⟩ ≡
#include "abacus/broadcast.h"
#include "abacus/master.h"
#include "abacus/parmaster.h"
#include <ace/OS.h>
#include <ace/SOCK_Stream.h>
  ACE_Thread_Mutex ABA_BROADCAST::mutex_;      /* instantiate the static members */
  int ABA_BROADCAST::broadcastit(ABA_BROADCAST *bcast, const char *buf, int len)
  {     /* encode the length */
    char buf_len[4];
    XDR xdr;
    xdrmem_create(&xdr, buf_len, 4, XDR_ENCODE);
    xdr_int(&xdr, &len);
    xdr_destroy(&xdr);      /* send messages to all hosts except myself */
    for (int i = 0; i < bcast→parmaster_→hostCount(); i++) {
      if (¬bcast→includeMe_ ∧ i ≡ bcast→parmaster_→hostId()) continue;      /* send the length */
      int r = bcast→parmaster_→notifyStream(i).send_n(buf_len, 4, 0);
      if (r ≠ 4) {
        ACE_ERROR((LM_ERROR, "%p\n", "Error in send_n"));
        assert(0);
      }     /* send the data */
      r = bcast→parmaster_→notifyStream(i).send_n(buf, len, 0);
      if (r ≠ len) {
        ACE_ERROR((LM_ERROR, "%p\n", "Error in send_n"));
        assert(0);
      }
    }
    return len;
  }
  int ABA_BROADCAST::broadcastreply(ABA_BROADCAST *bcast, char *buf, int len){
      /* the accumulated reply */
    bcast→reply_ = 0;      /* receive replies from all hosts except myself */
    int len1 = len; for (int i = 0; i < bcast→parmaster_→hostCount(); i++) {
    if (¬bcast→includeMe_ ∧ i ≡ bcast→parmaster_→hostId()) continue;      /* receive the length len1 */
    XDR xdr;
    char buf_len1[4];
    int r = bcast→parmaster_→notifyStream(i).recv_n(buf_len1, 4, 0);
    if (r ≠ 4) {
      ACE_ERROR((LM_ERROR, "%p\n", "broadcastreply: error in recv_n"));
      assert(0);
    }
    xdrmem_create(&xdr, buf_len1, 4, XDR_DECODE);
    xdr_int(&xdr, &len1);
    xdr_destroy(&xdr);
    assert(len1 ≤ len);
    assert(len1 ≥ 8);      /* at least the separator and one integer! */
      /* receive the reply (one integer) */
    r = bcast→parmaster_→notifyStream(i).recv_n(buf, len1, 0);
    if (r ≠ len1) {
```

```
    ACE_ERROR((LM_ERROR, "%p\n", "broadcastreply: error in recv_n"));
    assert(0);
  }     /* skip the separator and unpack one boolean into reply */
  bool reply;
  xdrmem_create(&xdr, buf + 4, len1 - 4, XDR_DECODE);
  assert(sizeof(bool) ≡ sizeof (bool_t)); r = xdr_bool (&xdr, ( bool_t * ) &reply ) ;
  assert(r);
  xdr_destroy(&xdr);      /* accumulate the reply */
  if (reply) bcast→reply_++;
  } return len1; }
```

See also sections 59, 60, 61, and 62.

59. The constructor.

Arguments:
master
A pointer to the master.

```
⟨ broadcast.cc 58 ⟩ +≡
ABA_BROADCAST::ABA_BROADCAST(const ABA_MASTER *master, bool includeMe):
  ABA_MESSAGEBASE((int(*)(void *, char *, int))broadcastreply, (int(*)(void *, char
      *, int))broadcastit),
  parmaster_(master→parmaster()),
  includeMe_(includeMe)
  {
    mutex_.acquire();
  }
```

60. Another constructor (protected).

```
⟨ broadcast.cc 58 ⟩ +≡
ABA_BROADCAST::ABA_BROADCAST(const ABA_MASTER *master, int(*readit)(void
      *, char *, int), int(*writeit)(void *, char *, int)):
  ABA_MESSAGEBASE(readit, writeit),
  parmaster_(master→parmaster()),
  includeMe_(false)
  {
    mutex_.acquire();
  }
```

61. The destructor.

```
⟨ broadcast.cc 58 ⟩ +≡
  ABA_BROADCAST::~ABA_BROADCAST()
  {
    mutex_.release();
  }
```

62. The function *receiveReply*( ) receives a reply of one boolean from every host in the system. The member variable *reply_* is set to the number of hosts which sent true.

Return Value:
The number of hosts which replied true.

⟨ broadcast.cc 58 ⟩ +≡

```
int ABA_BROADCAST::receiveReply()
{
  receive();      /* the following code is needed to trigger a call to */
    /* the function broadcastreply() which in turn receives a */
    /* boolean from each host and sets reply_. */
  bool dummy;
  unpack(dummy);
  return reply_;
}
```

63. NOTIFICATION MESSAGE.

The **ABA_NOTIFICATION** class is used to send short notification messages over one of the established notification streams of **ABA_PARMASTER**. These messages are handled by the notification server in the same way as broadcasts are. The difference to **ABA_BROADCAST** is, that a notification message is sent to a single host, only.

```
⟨ notification.h 63 ⟩ ≡
#ifndef ABA_NOTIFICATION_H
#define ABA_NOTIFICATION_H
#include "abacus/broadcast.h"
#include <ace/SOCK_Stream.h>
  class ABA_NOTIFICATION : public ABA_BROADCAST
  {
  public:
    ABA_NOTIFICATION(const ABA_MASTER *master, int dest);
  protected:
    static int readit(ABA_NOTIFICATION *msg, char *buf, int len);
    static int writeit(ABA_NOTIFICATION *msg, const char *buf, int len);
    ACE_SOCK_Stream *stream_;
  }
  ;
#endif    /* ¬ABA_NOTIFICATION_H */
```

64. The member functions are implemented in the file `notification.cc`.

```
⟨ notification.cc 64 ⟩ ≡
#include "abacus/notification.h"
#include "abacus/master.h"
#include "abacus/parmaster.h"
```

See also sections 65 and 66.

65. Static helper functions to read/write data from/to the stream

```
⟨ notification.cc 64 ⟩ +≡
  int ABA_NOTIFICATION::readit(ABA_NOTIFICATION *msg, char *buf, int len)
  {    /* receive and decode len1 (the length of the message) */
    XDR xdr;
    int len1;
    char buf_len1[4];
    int r = msg→stream_→recv_n(buf_len1, 4, 0);
    if (r ≠ 4) {
      ACE_ERROR((LM_ERROR, "%p\n", "Error in recv_n"));
      assert(0);
    }
    xdrmem_create(&xdr, buf_len1, 4, XDR_DECODE);
    xdr_int(&xdr, &len1);
    xdr_destroy(&xdr);
    assert(len1 ≤ len);      /* receive len1 bytes of data */
    r = msg→stream_→recv_n(buf, len1, 0);
    if (r ≠ len1) {
      ACE_ERROR((LM_ERROR, "%p\n", "Error in recv_n"));
      assert(0);
    }
    return r;
  }
  int ABA_NOTIFICATION::writeit(ABA_NOTIFICATION *msg, const char *buf, int len)
  {    /* encode and send the length */
    char buf_len[4];
    XDR xdr;
    xdrmem_create(&xdr, buf_len, 4, XDR_ENCODE);
    xdr_int(&xdr, &len);
    xdr_destroy(&xdr);
    int r = msg→stream_→send_n(buf_len, 4, 0);
    if (r ≠ 4) {
      ACE_ERROR((LM_ERROR, "%p\n", "Error in send_n"));
      assert(0);
    }    /* send the data */
    r = msg→stream_→send_n(buf, len, 0);
    if (r ≠ len) {
      ACE_ERROR((LM_ERROR, "%p\n", "Error in send_n"));
      assert(0);
    }
    return r;
  }
```

66. The constructor. A message buffer encapsulates a bidirectional communication stream. The underlaying ACE°_SOCK°_*Stream* has to be initialized later by using the member function *stream*( ).

```
⟨ notification.cc 64 ⟩ +≡
ABA_NOTIFICATION::ABA_NOTIFICATION(const ABA_MASTER *master, int i):
  ABA_BROADCAST(master, (int(*)(void *, char *, int))readit, (int(*)(void *, char *, int))writeit)
  {
    stream_ = &master→parmaster( )→notifyStream(i);
  }
```

67. HELPER CLASSES.

68. IDENTIFICATION OF OBJECTS.

An instance of the class **ABA_ID** is used for identification of the same object stored multiple times on different processors. It consists of a processor number, a sequence number, and optionally an index of a pool if a constraint/variable is identified. This class is used by the class **ABA_IDMAP**.

```
⟨ id.h 68 ⟩ ≡
#ifndef ABA_ID_H
#define ABA_ID_H
#include <iostream.h>
#include "abacus/abacusroot.h"
  class ABA_MESSAGE;
  class ABA_ID : public ABA_ABACUSROOT {
  public:
    ABA_ID();
    ABA_ID(ABA_MESSAGE &msg);
    void pack(ABA_MESSAGE &msg) const;
    friend ostream&operator≪(ostream & out,const ABA_ID &id);
    friend int operator≡(const ABA_ID &lhs,const ABA_ID &rhs);
    void initialize(unsigned long sequence,int proc,int index);
    void uninitialize();
    bool isInitialized() const;
    unsigned long sequence() const;
    int proc() const;
    int index() const;
  private:
    unsigned long sequence_;
    short proc_;
    short index_;
  };
  inline unsigned long ABA_ID::sequence() const
  {
    return sequence_;
  }
  inline int ABA_ID::proc() const
  {
    return proc_;
  }
  inline int ABA_ID::index() const
  {
    return index_;
  }
#endif     /* ¬ABA_ID_H */
```

69. The member functions are implemented in the file `id.cc`.

```
⟨ id.cc 69 ⟩ ≡
#include "abacus/id.h"
#include "abacus/message.h"
```

See also sections 70, 71, 72, 73, 74, 75, 76, and 77.

70. The constructor. *sequence_* will be set to 0 indicating that the **ABA_ID** is not initialized.

```
⟨ id.cc 69 ⟩ +≡
ABA_ID::ABA_ID():
  sequence_(0)
  {}
```

71. The message constructor.

Arguments:

msg
The message from which the object is initialized.

```
⟨ id.cc 69 ⟩ +≡
  ABA_ID::ABA_ID(ABA_MESSAGE &msg)
  {
    msg.unpack(sequence_);
    msg.unpack(proc_);
    msg.unpack(index_);
  }
```

72. The function *pack*() packs the **ABA_ID** in an **ABA_MESSAGE** object.

Arguments:

msg
The **ABA_MESSAGE** object in which the **ABA_ID** is packed.

```
⟨ id.cc 69 ⟩ +≡
  void ABA_ID::pack(ABA_MESSAGE &msg) const
  {
    msg.pack(sequence_);
    msg.pack(proc_);
    msg.pack(index_);
  }
```

73. The output operator.

Return Value:

A reference to the output stream.

Arguments:

out
The output stream.

id
The **ABA_ID** being output.

```
⟨ id.cc 69 ⟩ +≡
  ostream & operator≪(ostream & out, const ABA_ID &id)
  {
    if (id.sequence_ ≡ 0) return out ≪ "(uninitialized␣id)";
    else return out ≪ '(' ≪ id.sequence_ ≪ ',' ≪ id.proc_ ≪ ',' ≪ id.index_ ≪ ')';
  }
```

74. The comparison operator.

Return Value:

0
: If both **ABA_ID** objects are not equal,

1
: otherwise.

Arguments:

lhs
: The left hand side of the comparison.

rhs
: The right hand side of the comparison.

```
⟨ id.cc 69 ⟩ +≡
  int operator≡(const ABA_ID &lhs, const ABA_ID &rhs)
  {
    return lhs.sequence_ ≡ rhs.sequence_ ∧ lhs.proc_ ≡ rhs.proc_ ∧ lhs.index_ ≡ rhs.index_;
  }
```

75. The function *initialize*( ) sets the processor number and the sequence number.

Arguments:

sequence
: The sequence number of the **ABA_ID**.

proc
: The processor number of the **ABA_ID**.

index
: If a constraint/variable is identified, then this optional parameter determines the index of the pool in which the constraint/variable is stored.

```
⟨ id.cc 69 ⟩ +≡
  void ABA_ID::initialize(unsigned long sequence, int proc, int index)
  {
    sequence_ = sequence;
    proc_ = proc;
    index_ = index;
  }
```

76. The function *uninitialize*( ) sets the **ABA_ID** to the uninitialized state.

```
⟨ id.cc 69 ⟩ +≡
  void ABA_ID::uninitialize( )
  {
    sequence_ = 0;
  }
```

77. The function *isInitialized*( ) tests if the **ABA_ID** was initialized.

Return Value:

false
: If the **ABA_ID** is not initialized.

true
: If the **ABA_ID** is initialized.

⟨ id.cc 69 ⟩ +≡

```
bool ABA_ID::isInitialized( ) const
{
  return sequence_ ≠ 0;
}
```

78. IDENTIFICATION MAP.

The class **ABA_IDMAP**⟨*Type*⟩ implements a map between elements of **ABA_ID** and pointers to objects of the class *Type*. Objects beeing inserted are not copied, only a pointer to the object is stored in the map. Insertion and deletion of map elements need constant time in the average, since in the implementation a hashtable is used.

It is possible to create aliases of **ABA_ID** elements which are linked in a ringlist. If an object is removed from the map all **ABA_ID** elements referencing this object will be removed efficiently.

```
⟨ idmap.h 78 ⟩ ≡
#ifndef ABA_IDMAP_H
#define ABA_IDMAP_H
#include <ace/Synch.h>
#include "abacus/abacusroot.h"
#include "abacus/id.h"
  class ABA_MASTER;
  template⟨class Type⟩ class ABA_IDMAP : public ABA_ABACUSROOT {
public:
  ABA_IDMAP(const ABA_MASTER *master, int size, int index);
  ~ABA_IDMAP();
#ifdef ABACUS_NEW_TEMPLATE_SYNTAX
  friend ostream&operator≪ <> (ostream & out, const ABA_IDMAP⟨Type⟩ &idmap);
#else
  friend ostream&operator≪(ostream & out, const ABA_IDMAP⟨Type⟩ &idmap);
#endif
  Type * find(const ABA_ID &id);
  void insert(const ABA_ID &id, const Type*obj);
  void insertWithNewId(ABA_ID &id, const Type*obj);
  void insertAlias(const ABA_ID &id, const ABA_ID &aliasId);
  int remove(const ABA_ID &id);
  unsigned long sequence() const;
  int proc() const;
  int index() const;
private:
  const ABA_MASTER *master_; ABA_HASH < ABA_ID , const Type* > map_;
  unsigned long sequence_;
  int proc_;
  int index_;
  ACE_Thread_Mutex mp_; } ;
#include "abacus/idmap.inc"
  template⟨class Type⟩ inline unsigned long ABA_IDMAP⟨Type⟩::sequence() const
  {
    return sequence_;
  }
  ;
  template⟨class Type⟩ inline int ABA_IDMAP⟨Type⟩::proc() const
  {
    return proc_;
  }
  ;
  template⟨class Type⟩ inline int ABA_IDMAP⟨Type⟩::index() const
```

```
  {
    return index_;
  }
  ;
#endif     /* ¬ABA_IDMAP_H */
```

79.

```
⟨ idmap.inc 79 ⟩ ≡
#include "abacus/master.h"
#include "abacus/id.h"
#include "abacus/hash.h"
```

See also sections 80, 81, 82, 83, 84, 85, and 86.

80. The constructor.

Arguments:

master
A pointer to the corresponding master object.

size
The size of the hashtable used by the map.

pool
The index of the pool, if the **ABA_ID** identifies a constraint/variable.

```
⟨ idmap.inc 79 ⟩ +≡
template⟨class Type⟩ ABA_IDMAP⟨Type⟩::ABA_IDMAP(const ABA_MASTER *master, int
      size, int index): master_(master),
  map_(master, size),
  sequence_(1),
  index_(index)
  {
    proc_ = master→parmaster()→hostId();
  }
```

81. The destructor.

```
⟨ idmap.inc 79 ⟩ +≡
  template⟨class Type⟩ ABA_IDMAP⟨Type⟩::~ABA_IDMAP()
  {}
```

82. The output operator writes the **ABA_IDMAP** to the stream *out*.

Return Value:
A reference to the output stream.

Arguments:
out
The output stream.
idmap
The **ABA_IDMAP** being output.

```
⟨ idmap.inc 79 ⟩ +≡
  template⟨class Type⟩ ostream&operator≪(ostream & out, const ABA_IDMAP⟨Type⟩ &idmap)
  {
    idmap.mp_.acquire();
    out ≪ "ABA_IDMAP:␣proc=" ≪ idmap.proc_ ≪ ",␣sequence=" ≪ idmap.sequence_ ≪ endl;
    out ≪ idmap.map_;
    idmap.mp_.release();
    return out;
  }
```

83. The function *find*() looks up an object in the map by its **ABA_ID**.

Return Value:
A pointer to the object with the given **ABA_ID**, or a 0-pointer if there is no such object.

Arguments:
id
The **ABA_ID** of the object beeing looked up.

```
⟨ idmap.inc 79 ⟩ +≡
  template⟨class Type⟩ Type*ABA_IDMAP⟨Type⟩::find(const ABA_ID &id){ mp_.acquire();
        const Type**ptr = map_.find(id);
        mp_.release(); return ptr ? ( Type * )
      (*ptr) : 0; }
```

84. The function *insert*( ).

Arguments:

id

The identification of the object.

obj

A pointer to the object beeing inserted.

```
⟨ idmap.inc 79 ⟩ +≡
  template⟨class Type⟩ void ABA_IDMAP⟨Type⟩::insert(const ABA_ID &id, const Type*obj)
  {
    mp_.acquire( );
#ifdef ABACUSSAFE
    if (map_.find(id)) {
      master_→err( ) ≪ "ABA_IDMAP::insert():␣tried␣to␣insert␣ABA_ID␣" ≪ id ≪
          "␣more␣than␣once.";
      exit(Fatal);
    }
#endif
    map_.insert(id, obj);
    mp_.release( );
  }
```

85. The function *insertWithNewId*( ) inserts the object after having assigned a new **ABA_ID** to it. The new **ABA_ID** consists of the processor number and a sequence number which is incremented each time this function is called.

Arguments:

id

The new identification is assigned to *id*.

obj

A pointer to the object beeing inserted.

```
⟨ idmap.inc 79 ⟩ +≡
  extern "C"
  {
#include <limits.h>     /* ULONG_MAX */
  }
  template⟨class Type⟩ void ABA_IDMAP⟨Type⟩::insertWithNewId(ABA_ID &id, const Type*obj)
  {
    mp_.acquire();
    if (sequence_ ≡ ULONG_MAX) {
      master_→err() ≪ "ABA_IDMAP::insertWithNewId():␣insertion\
          ␣failed,␣""maximum␣sequence␣number␣ULONG_MAX␣=␣" ≪ ULONG_MAX ≪ "␣reached";
      exit(Fatal);
    }
    id.initialize(sequence_++, proc_, index_);
#ifdef ABACUSSAFE
    if (map_.find(id)) {
      master_→err() ≪ "ABA_IDMAP::insertWithNewId():␣tried␣to␣insert␣ABA_ID␣" ≪ id ≪
          "␣more␣than␣once.";
      exit(Fatal);
    }
#endif
    map_.insert(id, obj);
    mp_.release();
  }
```

86. The function *remove*( ).

Return Value:

0

If the **ABA_ID** was successfully removed.

1

If there is no such **ABA_ID** in the map.

Arguments:

id

The identification beeing removed.

```
⟨ idmap.inc 79 ⟩ +≡
  template⟨class Type⟩ int ABA_IDMAP⟨Type⟩::remove(const ABA_ID &id)
  {
    mp_.acquire();
    int status = map_.remove(id);
    mp_.release();
    return status;
  }
```

87. INTEGER SET.

The class `ABA_INTSET` implements the abstract datatype for storing a subset of the set $\{0, \ldots, n-1\}$. All operations of this class need constant time only. The storage requirement is $O(n)$.

⟨ `intset.h` 87 ⟩ ≡

```
#ifndef ABA_INTSET_H
#define ABA_INTSET_H
  class ABA_INTSET {
  public:
    ABA_INTSET();
    ~ABA_INTSET();
    void initialize(int n);
    bool exists(int elem) const;
    void insert(int elem);
    void remove(int elem);
    int count() const;
    int elem(int index) const;
  private:
    int *uninitializedMap_;
    int *position_;
    int count_;
  };
#endif    /* ¬ABA_INTSET_H */
```

88. The member functions are defined in the file `intset.cc`.

The set is implemented by the arrays *uninitializedMap_*[] and *position_*[]. The following holds for each element e stored in the set.

a) $0 \leq uninitializedMap\_[e] < count\_$

b) $position\_[uninitializedMap\_[e]] = e$

It is not required to initialize the arrays.

⟨ `intset.cc` 88 ⟩ ≡

```
#include "abacus/intset.h"
```

See also sections 89, 90, 91, 92, 93, 94, 95, and 96.

89. The constructor. An empty set is created. $O(n)$ bytes of memory are allocated for the internal representation of the data structure.

Arguments:

n

The created object can store subsets of $\{0, \ldots, n-1\}$.

⟨ `intset.cc` 88 ⟩ +≡

```
ABA_INTSET::ABA_INTSET(): uninitializedMap_(0),
  position_(0),
  count_(0)
  { }
```

90. The destructor deletes the allocated memory.

```
⟨ intset.cc 88 ⟩ +≡
  ABA_INTSET::~ABA_INTSET()
  {
    delete []uninitializedMap_;
    delete []position_;
  }
```

91. An empty set is initialized. $O(n)$ bytes of memory are allocated for the internal representation of the data structure.

Arguments:

n

The initialized object can store subsets of $\{0, \ldots, n-1\}$.

```
⟨ intset.cc 88 ⟩ +≡
  void ABA_INTSET::initialize(int n)
  {
    delete []uninitializedMap_;
    delete []position_;
    uninitializedMap_ = newint[n];
    position_ = newint[n];
    count_ = 0;
  }
```

92. The function *exists*() tests if an integer is contained in the set.

```
⟨ intset.cc 88 ⟩ +≡
  bool ABA_INTSET::exists(int elem) const
  {
    int i = uninitializedMap_[elem];
    return (unsigned) i < (unsigned) count_ ∧ position_[i] ≡ elem;
  }
```

93. The function *insert*() inserts an element in the set. The element is not inserted twice if it is already contained in the set.

Arguments:

elem

The element to be inserted.

```
⟨ intset.cc 88 ⟩ +≡
  void ABA_INTSET::insert(int elem)
  {
    int i = uninitializedMap_[elem];
    if ((unsigned) i < (unsigned) count_ ∧ position_[i] ≡ elem) return;
        /* elem is not contained in the set. It is inserted by the next two lines. */
    uninitializedMap_[elem] = count_;
    position_[count_++] = elem;
  }
```

94. The function *remove*( ) removes an element from the set if it is contained in it.

Arguments:
elem
The element to be removed.

```
⟨ intset.cc 88 ⟩ +≡
  void ABA_INTSET::remove(int elem)
  {
    int i = uninitializedMap_[elem];
    if ((unsigned) i < (unsigned) count_ ∧ position_[i] ≡ elem) {
        /* elem is contained in the set. We have to remove it. */
      if (--count_ ≠ i) {
        int help = position_[count_];
        position_[i] = help;
        uninitializedMap_[help] = i;
      }
    }
  }
```

95. The function *count*( ). This function and the function *elem*( ) can be used to iterate over the elements of the set.

Return Value:
The number of elements contained in the set.

```
⟨ intset.cc 88 ⟩ +≡
  int ABA_INTSET::count( ) const
  {
    return count_;
  }
```

96. The function *elem*( ) is used to access the elements contained in the set.

Arguments:
index
The index of the element to be returned. This must be a non negative integer less than *number*( ).

Return Value:
The element referenced by the given index.

```
⟨ intset.cc 88 ⟩ +≡
  int ABA_INTSET::elem(int index) const
  {
    return position_[index];
  }
```

97. DUAL BOUND.

The class **ABA_DUALBOUND** implements the abstract datatype for holding up to n dual bounds $\{d_1, \ldots, d_n\}$. A dual bound d_i can be inserted or removed. The best bound (minimum or maximum) can be queried.

```
⟨ dualbound.h 97 ⟩ ≡
#ifndef ABA_DUALBOUND_H
#define ABA_DUALBOUND_H
#include "abacus/intset.h"
  class ABA_GLOBAL;
  class ABA_DUALBOUND {
  public:
    ABA_DUALBOUND(ABA_GLOBAL *glob);
    ~ABA_DUALBOUND();
    void initialize(int n, bool minIsBest);
    void insert(int i, double d);
    void remove(int i);
    bool better(int i, double d) const;
    double best(int *index = 0) const;
    double best(double d) const;
    double worst() const;
  private:
    void updateBestAndWorst();
    ABA_GLOBAL *glob_;
    ABA_INTSET set_;
    double *bounds_;
    double best_;
    int bestIndex_;
    double worst_;
    bool minIsBest_;
  };
  inline double ABA_DUALBOUND::best(int *index) const
  {
    if (index) *index = bestIndex_;
    return best_;
  }
  inline double ABA_DUALBOUND::worst() const
  {
    return worst_;
  }
#endif     /* ¬ABA_DUALBOUND_H */
```

98. The member functions are defined in the file `dualbound.cc`.

```
⟨ dualbound.cc 98 ⟩ ≡
#include "abacus/dualbound.h"
#include "abacus/global.h"
```

See also sections 99, 100, 101, 102, 103, 104, 106, and 108.

99. The constructor.

Arguments:
glob
A pointer to the corresponding global object.

```
⟨ dualbound.cc 98 ⟩ +≡
ABA_DUALBOUND::ABA_DUALBOUND(ABA_GLOBAL *glob):
  glob_(glob),
  bounds_(0)
  { }
```

100. The destructor deletes the allocated memory.

```
⟨ dualbound.cc 98 ⟩ +≡
  ABA_DUALBOUND::~ABA_DUALBOUND()
  {
    delete []bounds_;
  }
```

101. This function initializes the set of dual bounds.

Arguments:
n
The object can hold up to n dual bounds.
minIsBest
If this parameter is true the function *best*() returns the minimum of the dual bounds, otherwise it returns the maximum.

```
⟨ dualbound.cc 98 ⟩ +≡
  void ABA_DUALBOUND::initialize(int n, bool minIsBest)
  {
    minIsBest_ = minIsBest;
    set_.initialize(n);
    delete []bounds_;
    bounds_ = newdouble[n];
    updateBestAndWorst();
  }
```

102. The function *insert*( ) inserts a dual bound in the set. If a bound with the same index already exists in the set the value of the bound is updated.

Arguments:

i

The index of the dual bound (0..n-1).

d

The dual bound.

```
⟨ dualbound.cc 98 ⟩ +≡
  void ABA_DUALBOUND::insert(int i, double d)
  {
    if (set_.exists(i) ∧ (bounds_[i] ≡ best_ ∨ bounds_[i] ≡ worst_)) {
      bounds_[i] = d;
      updateBestAndWorst( );
    }
    else {
      set_.insert(i);
      bounds_[i] = d;
      if (minIsBest_) {
        if (d < best_) {
          best_ = d;
          bestIndex_ = i;
        }
        if (d > worst_) worst_ = d;
      }
      else {
        if (d > best_) {
          best_ = d;
          bestIndex_ = i;
        }
        if (d < worst_) worst_ = d;
      }
    }
  }
```

103. The function *remove*( ) removes a dual bound from the set.

Arguments:

i

The index of the dual bound (0..n-1).

```
⟨ dualbound.cc 98 ⟩ +≡
  void ABA_DUALBOUND::remove(int i)
  {
    if (set_.exists(i)) {
      set_.remove(i);
      if (bounds_[i] ≡ best_ ∨ bounds_[i] ≡ worst_) updateBestAndWorst( );
    }
  }
```

104. The function *better*( ).

Arguments:

i

The index of the dual bound (0..n-1).

d

The value of the dual bound to be tested.

Return Value:

true

if d is a new dual bound or d is better than the previous dual bound at index i,

false

otherwise.

⟨ dualbound.cc 98 ⟩ +≡

```
bool ABA_DUALBOUND::better(int i, double d) const
{
  if (set_.exists(i)) {      /* test, if the new bound is better than the old one */
    double oldBound = bounds_[i];
    if (minIsBest_) {
      if (d ≤ oldBound) return false;
    }
    else {
      if (d ≥ oldBound) return false;
    }
  }
  return true;
}
```

105. The function *best*( ). Returns the best dual bound (minimum or maximum) of all dual bounds in the set.

Arguments:

index

An pointer to an integer to which the index of the best dual bound should be stored. This is an optional parameter.

Return Value:

The best dual bound.

double ABA_DUALBOUND::best(int *index) const

106. This version of the function *best*( ) returns the best dual bound (minimum or maximum) of all dual bounds in the set and a specified dual bound.

Arguments:

d

An additional dual bound.

Return Value:

The best dual bound.

```
⟨ dualbound.cc 98 ⟩ +≡
  double ABA_DUALBOUND::best(double d) const
  {
    if (minIsBest_) return d < best_ ? d : best_;
    else return d > best_ ? d : best_;
  }
```

107. The function *worst*( ). Returns the worst dual bound (minimum or maximum) of all dual bounds in the set.

Return Value:

The worst dual bound.

double ABA_DUALBOUND::worst() const

108. The function *updateBestAndWorst*( ) updates the variables *best_*, *worst_* and *bestIndex_*.

```
⟨ dualbound.cc 98 ⟩ +≡
  void ABA_DUALBOUND::updateBestAndWorst()
  {
    bestIndex_ = −1;
    if (minIsBest_) {
      int count = set_.count();
      best_ = glob_→infinity();
      worst_ = −glob_→infinity();
      for (int i = 0; i < count; i++) {
        double d = bounds_[set_.elem(i)];
        if (d < best_) {
          best_ = d;
          bestIndex_ = set_.elem(i);
        }
        if (d > worst_) worst_ = d;
      }
    }
    else {
      int count = set_.count();
      best_ = −glob_→infinity();
      worst_ = glob_→infinity();
      for (int i = 0; i < count; i++) {
        double d = bounds_[set_.elem(i)];
        if (d > best_) {
          best_ = d;
          bestIndex_ = set_.elem(i);
        }
        if (d < worst_) worst_ = d;
      }
    }
  }
```

109. SERIALIZATION.

This section contains extensions for serialization of several ABACUS classes.

110. ABA_CSENSE.

111. The message constructor creates the **ABA_CSENSE** from an **ABA_MESSAGE**.

Arguments:

msg
The **ABA_MESSAGE** object from which the sense is initialized.

```
⟨ csense.cc 111 ⟩ ≡
ABA_CSENSE::ABA_CSENSE(const ABA_GLOBAL *glob, ABA_MESSAGE &msg):
  glob_(glob)
  {
    sense_ = (SENSE)msg.unpackInt( );
  }
```

See also section 112.

112. The function *pack*( ) packs the data of the sense in an **ABA_MESSAGE** object.

Arguments:

msg
The **ABA_MESSAGE** object in which the sense is packed.

```
⟨ csense.cc 111 ⟩ +≡
  void ABA_CSENSE::pack(ABA_MESSAGE &msg) const
  {
    msg.pack((int) sense_);
  }
```

113. ABA_VARTYPE.

114. The message constructor creates the type from an **ABA_MESSAGE**.

Arguments:

msg

The message from which the type is initialized.

```
⟨ vartype.cc 114 ⟩ ≡
  ABA_VARTYPE::ABA_VARTYPE(ABA_MESSAGE &msg)
  {
    type_ = (TYPE)msg.unpackInt();
  }
```

See also section 115.

115. The function *pack*( ) packs the type in an **ABA_MESSAGE**.

Arguments:

msg

The **ABA_MESSAGE** object in which the type is packed.

```
⟨ vartype.cc 114 ⟩ +≡
  void ABA_VARTYPE::pack(ABA_MESSAGE &msg) const
  {
    msg.pack((int) type_);
  }
```

116. ABA_FSVARSTAT.

117. The message constructor creates the status from an **ABA_MESSAGE**.

Arguments:

glob
: A pointer to a global object.

msg
: The message from which the object is initialized.

```
⟨ fsvarstat.cc 117 ⟩ ≡
ABA_FSVARSTAT::ABA_FSVARSTAT(const ABA_GLOBAL *glob, ABA_MESSAGE &msg):
  glob_(glob)
  {
    status_ = (STATUS) msg.unpackInt();
    msg.unpack(value_);
  }
```

See also section 118.

118. The function *pack*( ) packs the data of the status in an **ABA_MESSAGE**.

Arguments:

msg
: The **ABA_MESSAGE** object in which the status is packed.

```
⟨ fsvarstat.cc 117 ⟩ +≡
  void ABA_FSVARSTAT::pack(ABA_MESSAGE &msg) const
  {
    msg.pack((int) status_);
    msg.pack(value_);
  }
```

119. ABA_LPVARSTAT.

120. The message constructor creates the **ABA_LPVARSTAT** from a message.

Arguments:

glob
A pointer to a global object.

msg
The message from which the object is initialized.

```
⟨ lpvarstat.cc 120 ⟩ ≡
ABA_LPVARSTAT::ABA_LPVARSTAT(const ABA_GLOBAL *glob, ABA_MESSAGE &msg):
  glob_(glob)
  {
    status_ = (STATUS) msg.unpackInt();
  }
```

See also section 121.

121. The function *pack*( ) packs the data of the **ABA_LPVARSTAT** in a message.

Arguments:

msg
The **ABA_MESSAGE** object in which the status is packed.

```
⟨ lpvarstat.cc 120 ⟩ +≡
  void ABA_LPVARSTAT::pack(ABA_MESSAGE &msg) const
  {
    msg.pack((int) status_);
  }
```

122. ABA_SLACKSTAT.

123. The message constructor creates the status from a message.

Arguments:

glob
: A pointer to a global object.

msg
: The message from which the object is initialized.

```
⟨ slackstat.cc 123 ⟩ ≡
ABA_SLACKSTAT::ABA_SLACKSTAT(const ABA_GLOBAL *glob, ABA_MESSAGE &msg):
  glob_(glob)
  {
    status_ = (STATUS) msg.unpackInt();
  }
```

See also section 124.

124. The function *pack*( ) packs the data of the status in a message.

Arguments:

msg
: The **ABA_MESSAGE** object in which the status is packed.

```
⟨ slackstat.cc 123 ⟩ +≡
  void ABA_SLACKSTAT::pack(ABA_MESSAGE &msg) const
  {
    msg.pack((int) status_);
  }
```

125. ABA_BRANCHRULE.

⟨ branchrule.h 125 ⟩ ≡
```
#define SETBRANCHRULE_CLASSID 9001    /* preliminary */
#define VALBRANCHRULE_CLASSID 9002    /* preliminary */
#define CONBRANCHRULE_CLASSID 9003    /* preliminary */
#define BOUNDBRANCHRULE_CLASSID 9004    /* preliminary */
```

126. The message constructor creates the **ABA_BRANCHRULE** from an **ABA_MESSAGE**.

Arguments:

master

A pointer to the corresponding master of the optimization.

msg

The message from which the object is initialized.

⟨ branchrule.cc 126 ⟩ ≡
```
ABA_BRANCHRULE::ABA_BRANCHRULE(const ABA_MASTER *master,
      ABA_MESSAGE &msg):
  master_(master)
  { }
```
See also section 127.

127. The function *pack*( ) packs the data of the **ABA_BRANCHRULE** in an **ABA_MESSAGE** object. This virtual function has to be redefined in a derived class if additional data should be packed. In this case the *pack*( ) function of the base class must be called by the *pack*( ) function of the derived class.

Arguments:

msg

The **ABA_MESSAGE** object in which the **ABA_BRANCHRULE** is packed.

⟨ branchrule.cc 126 ⟩ +≡
```
void ABA_BRANCHRULE::pack(ABA_MESSAGE &msg) const
  { }
```

128. ABA_SETBRANCHRULE.

129. The message constructor creates the **ABA_SETBRANCHRULE** from an **ABA_MESSAGE**.

Arguments:

master
A pointer to the corresponding master of the optimization.

msg
The message from which the object is initialized.

```
⟨ setbranchrule.cc 129 ⟩ ≡
ABA_SETBRANCHRULE::ABA_SETBRANCHRULE(const ABA_MASTER
    *master, ABA_MESSAGE &msg):
  ABA_BRANCHRULE(master, msg)
  {
    msg.unpack(variable_);
    status_ = (ABA_FSVARSTAT::STATUS) msg.unpackInt();
    msg.unpack(oldLpBound_);
  }
```

See also sections 130 and 131.

130. The virtual function *pack*( ) packs the data of the **ABA_SETBRANCHRULE** in an **ABA_MESSAGE** object. The *pack*( ) function of the base class has to be called before the local data is packed.

Arguments:

msg
The **ABA_MESSAGE** object in which the **ABA_SETBRANCHRULE** is packed.

```
⟨ setbranchrule.cc 129 ⟩ +≡
  void ABA_SETBRANCHRULE::pack(ABA_MESSAGE &msg) const
  {
    ABA_BRANCHRULE::pack(msg);
    msg.pack(variable_);
    msg.pack((int) status_);
    msg.pack(oldLpBound_);
  }
```

131. The abstract virtual function *classId*( ) of **ABA_BRANCHRULE** has to be implemented in each derived subclass. It has to return an integer which identifies the class.

Return Value:

The class identification.

```
⟨ setbranchrule.cc 129 ⟩ +≡
  int ABA_SETBRANCHRULE::classId() const
  {
    return SETBRANCHRULE_CLASSID;
  }
```

132. ABA_BOUNDBRANCHRULE.

133. The message constructor creates the **ABA_BOUNDBRANCHRULE** from an **ABA_MESSAGE**.

Arguments:

master

A pointer to the corresponding master of the optimization.

msg

The message from which the object is initialized.

```
⟨ boundbranchrule.cc 133 ⟩ ≡
ABA_BOUNDBRANCHRULE::ABA_BOUNDBRANCHRULE(const ABA_MASTER
    *master, ABA_MESSAGE &msg):
  ABA_BRANCHRULE(master, msg)
  {
    msg.unpack(variable_);
    msg.unpack(lBound_);
    msg.unpack(uBound_);
    msg.unpack(oldLpLBound_);
    msg.unpack(oldLpUBound_);
  }
```

See also sections 134 and 135.

134. The virtual function *pack*( ) packs the data of the **ABA_BOUNDBRANCHRULE** in an **ABA_MESSAGE** object. The *pack*( ) function of the base class has to be called before the local data is packed.

Arguments:

msg

The **ABA_MESSAGE** object in which the **ABA_BOUNDBRANCHRULE** is packed.

```
⟨ boundbranchrule.cc 133 ⟩ +≡
  void ABA_BOUNDBRANCHRULE::pack(ABA_MESSAGE &msg) const
  {
    ABA_BRANCHRULE::pack(msg);
    msg.pack(variable_);
    msg.pack(lBound_);
    msg.pack(uBound_);
    msg.pack(oldLpLBound_);
    msg.pack(oldLpUBound_);
  }
```

135. The abstract virtual function *classId*( ) of **ABA_BRANCHRULE** has to be implemented in each derived subclass. It has to return an integer which identifies the class.

Return Value:

The class identification.

```
⟨ boundbranchrule.cc 133 ⟩ +≡
  int ABA_BOUNDBRANCHRULE::classId( ) const
  {
    return BOUNDBRANCHRULE_CLASSID;
  }
```

136. ABA_VALBRANCHRULE.

137. The message constructor creates the **ABA_VALBRANCHRULE** from an **ABA_MESSAGE**.

Arguments:

master
A pointer to the corresponding master of the optimization.

msg
The message from which the object is initialized.

```
⟨ valbranchrule.cc 137 ⟩ ≡
ABA_VALBRANCHRULE::ABA_VALBRANCHRULE(const ABA_MASTER
    *master, ABA_MESSAGE &msg):
  ABA_BRANCHRULE(master, msg)
  {
    msg.unpack(variable_);
    msg.unpack(value_);
    msg.unpack(oldLpLBound_);
    msg.unpack(oldLpUBound_);
  }
```

See also sections 138 and 139.

138. The virtual function *pack*( ) packs the data of the **ABA_VALBRANCHRULE** in an **ABA_MESSAGE** object. The *pack*( ) function of the base class has to be called before the local data is packed.

Arguments:

msg
The **ABA_MESSAGE** object in which the **ABA_VALBRANCHRULE** is packed.

```
⟨ valbranchrule.cc 137 ⟩ +≡
  void ABA_VALBRANCHRULE::pack(ABA_MESSAGE &msg) const
  {
    ABA_BRANCHRULE::pack(msg);
    msg.pack(variable_);
    msg.pack(value_);
    msg.pack(oldLpLBound_);
    msg.pack(oldLpUBound_);
  }
```

139. The abstract virtual function *classId*( ) of **ABA_BRANCHRULE** has to be implemented in each derived subclass. It has to return an integer which identifies the class.

Return Value:

The class identification.

```
⟨ valbranchrule.cc 137 ⟩ +≡
  int ABA_VALBRANCHRULE::classId( ) const
  {
    return VALBRANCHRULE_CLASSID;
  }
```

140. ABA_CONBRANCHRULE.

141. The message constructor creates the **ABA_CONBRANCHRULE** from an **ABA_MESSAGE**.

Arguments:

master
A pointer to the corresponding master of the optimization.

msg
The message from which the object is initialized.

```
⟨ conbranchrule.cc 141 ⟩ ≡
ABA_CONBRANCHRULE::ABA_CONBRANCHRULE(const ABA_MASTER
    *master, ABA_MESSAGE &msg):
  ABA_BRANCHRULE(master, msg),
  poolSlotRef_(master)
  {
    ABA_ID id(msg);
    char needConstraint;
    ABA_POOL⟨ABA_CONSTRAINT,
        ABA_VARIABLE⟩ *pool = (ABA_POOL⟨ABA_CONSTRAINT, ABA_VARIABLE⟩ *)
        master->parmaster()->getPool(id.index());
    ABA_POOLSLOT⟨ABA_CONSTRAINT, ABA_VARIABLE⟩ *ps = pool->findSlot(id);
    if (ps) {
      poolSlotRef_.slot(ps);
      needConstraint = 0;
    }
    else needConstraint = 1;      /* msg.clear(); */
    msg.pack(needConstraint);
    msg.send();      /* send flag if constraint is missing */
    msg.receive();      /* receive constraint if needed */
    if (needConstraint) {
      int classId;
      msg.unpack(classId);
      ABA_CONSTRAINT *con = (ABA_CONSTRAINT *) master->unpackConVar(msg, classId);
      if (debug(DEBUG_MESSAGE_CONVAR)) {
        master->out() ≪ "DEBUG_MESSAGE_CONVAR: Constraint " ≪ id ≪ " (classId=" ≪ classId ≪
            ") of ABA_CONBRANCHRULE received." ≪ endl;
      }
      ABA_POOL⟨ABA_CONSTRAINT,
          ABA_VARIABLE⟩ *pool = (ABA_POOL⟨ABA_CONSTRAINT, ABA_VARIABLE⟩
          *) master->parmaster()->getPool(id.index());
      ps = pool->insert(con);
      if (ps ≡ 0) {
        master->err() ≪ "ABA_CONBRANCHRULE::ABA_CONBRANCHRULE():\
            no room""to insert constraint into pool." ≪ endl;
        exit(Fatal);
      }
      ps->setIdentification(id);
      poolSlotRef_.slot(ps);
    }
  }
```

See also sections 142 and 143.

142. The virtual function *pack*( ) packs the data of the **ABA_CONBRANCHRULE** in an **ABA_MESSAGE** object. The *pack*( ) function of the base class has to be called before the local data is packed.

Arguments:

msg

The **ABA_MESSAGE** object in which the **ABA_CONBRANCHRULE** is packed.

```
⟨ conbranchrule.cc 141 ⟩ +≡
  void ABA_CONBRANCHRULE::pack(ABA_MESSAGE &msg) const
  {
    ABA_BRANCHRULE::pack(msg);
    if (poolSlotRef_.conVar( ) ≡ 0) {      /* removed conVar */
      master_→err( ) ≪ "ABA_CONBRANCHRULE::pack(): branching constraint not"" available!" ≪
          endl;
      exit(Fatal);
    }
    ABA_POOLSLOT⟨ABA_CONSTRAINT, ABA_VARIABLE⟩ *ps = poolSlotRef_.slot( );
    if (¬ps→getIdentification( ).isInitialized( )) ps→setNewIdentification( );
    ps→getIdentification( ).pack(msg);
    msg.send( );      /* send ABA_ID of constraint */
    msg.receive( );      /* receive flag if constraint is missing */
    char needConstraint;
    msg.unpack(needConstraint);      /* msg.clear(); */
    if (needConstraint) {
      ABA_CONSTRAINT *con = poolSlotRef_.conVar( );
      if (debug(DEBUG_MESSAGE_CONVAR)) {
        master_→out( ) ≪ "DEBUG_MESSAGE_CONVAR: sending Constraint " ≪ ps→getIdentification( ) ≪
            " (classId=" ≪ con→classId( ) ≪ ") of ABA_CONBRANCHRULE..." ≪ endl;
      }
      msg.pack(con→classId( ));
      con→pack(msg);
    }
  }
```

143. The abstract virtual function *classId*( ) of **ABA_BRANCHRULE** has to be implemented in each derived subclass. It has to return an integer which identifies the class.

Return Value:

The class identification.

```
⟨ conbranchrule.cc 141 ⟩ +≡
  int ABA_CONBRANCHRULE::classId( ) const
  {
    return CONBRANCHRULE_CLASSID;
  }
```

144. ABA_POOL.

145. The function *findSlot*( ) checks if a constraint/variable with the supplied **ABA_ID** exists in the pool.

Return Value:

A pointer to the poolslot of the constraint/variable if it exists in the pool, otherwise 0.

```
⟨ pool.inc 145 ⟩ ≡
  template⟨class BaseType, class CoType⟩ ABA_POOLSLOT⟨BaseType, CoType⟩
            *ABA_POOL⟨BaseType, CoType⟩::findSlot(const ABA_ID &id) const
      {
        return identificationMap_→find(id);
      }
```

See also section 146.

146. The function *identificationMap*( ).

Return Value:

A pointer to the **ABA_IDMAP** of the **ABA_POOL** of this processor.

```
⟨ pool.inc 145 ⟩ +≡
  template⟨class BaseType, class CoType⟩ ABA_IDMAP⟨ABA_POOLSLOT⟨BaseType, CoType⟩
            ␣⟩ *ABA_POOL⟨BaseType, CoType⟩::identificationMap( ) const
      {
        return identificationMap_;
      }
```

147. ABA_POOLSLOT.

148. The function *getIdentification*( ) returns the system wide identification of the constraint/variable referenced by the **ABA_POOLSLOT**.

Return Value:
The **ABA_ID** of the constraint/variable referenced by the **ABA_POOLSLOT**.

```
⟨poolslot.inc 148⟩ ≡
  template⟨class BaseType, class CoType⟩ const ABA_ID &ABA_POOLSLOT⟨BaseType,
          CoType⟩::getIdentification(void) const
    {
      if (conVar_ ≡ 0) {
        master_→err( ) ≪ "ABA_POOLSLOT::getIdentification()␣:␣no␣""constraint/variable\
            ␣available␣in␣this␣slot!" ≪ endl;
        exit(Fatal);
      }
      return conVar_→identification_;
    }
```

See also sections 149 and 150.

149. The function *setIdentification*( ) sets the system wide identification of the constraint/variable referenced by the **ABA_POOLSLOT** and registers it with the **ABA_IDMAP** of the pool.

Arguments:
id
The **ABA_ID** to be assigned.

```
⟨poolslot.inc 148⟩ +≡
  template⟨class BaseType, class CoType⟩ void ABA_POOLSLOT⟨BaseType,
          CoType⟩::setIdentification(const ABA_ID &id)
  {
    if (conVar_) {
      conVar_→identification_ = id;
      pool_→identificationMap( )→insert(id, this);
    }
    else {
      master_→err( ) ≪ "ABA_POOLSLOT::setIdentification()␣:␣no␣""constraint/variable\
          ␣available␣in␣this␣slot!" ≪ endl;
      exit(Fatal);
    }
  }
```

150. The function *setNewIdentification*( ) sets the system wide identification of the constraint/variable referenced by the **ABA_POOLSLOT** to a new **ABA_ID** and registers it with the **ABA_IDMAP** of the pool.

```
⟨ poolslot.inc 148 ⟩ +≡
  template⟨class BaseType, class CoType⟩ void ABA_POOLSLOT⟨BaseType,
          CoType⟩::setNewIdentification( )
  {
    if (conVar_) pool_→identificationMap( )→insertWithNewId(conVar_→identification_, this);
    else {
      master_→err( ) ≪ "ABA_POOLSLOT::setNewIdentification()␣:␣no␣""constraint/variable\
          ␣available␣in␣this␣slot!" ≪ endl;
      exit(Fatal);
    }
  }
```

151. ABA_CONVAR.

⟨ convar.h 151 ⟩ ≡
```
#define NUMCON_CLASSID 8001    /* preliminary */
#define ROWCON_CLASSID 8002    /* preliminary */
#define SROWCON_CLASSID 8003   /* preliminary */
#define NUMVAR_CLASSID 8004    /* preliminary */
#define COLVAR_CLASSID 8005    /* preliminary */
```

152. The message constructor creates the constraint/variable from an **ABA_MESSAGE**.

A constraint/variable can only be constructed by the message constructor, if the associated subproblem of the constraint/variable already exists in the memory of the receiving processor. This is verified using the **ABA_ID** of the subproblem. The *sub_* pointer of the constraint/variable is then set to this subproblem.

Arguments:

master
: A pointer to the corresponding master of the optimization.

msg
: The message from which the object is initialized.

⟨ convar.cc 152 ⟩ ≡
```
ABA_CONVAR::ABA_CONVAR(const ABA_MASTER *master, ABA_MESSAGE &msg):
  master_(master),
  sub_(0),
  expanded_(false),
  nReferences_(0),
  nActive_(0),
  nLocks_(0)
  {
    int bits = msg.unpackInt();
    assert((bits & 1) ≡ 0);      /* sub_ must be 0 */
    dynamic_ = ((bits & 2) ? true : false);
    local_ = ((bits & 4) ? true : false);
  }
```
See also section 153.

153. The function *pack*( ) packs the data of the constraint/variable in an **ABA_MESSAGE** object. This virtual function has to be redefined in a derived class if additional data should be packed. In this case the *pack*( ) function of the base class must be called by the *pack*( ) function of the derived class.

The constraint/variable must be packed and unpacked in compressed format. Note, that it is not required for the constraint/variable to be in compressed format at the time when it is packed!

Arguments:

msg
: The **ABA_MESSAGE** object in which the constraint/variable is packed.

⟨ convar.cc 152 ⟩ +≡
```
  void ABA_CONVAR::pack(ABA_MESSAGE &msg) const
  {
    msg.pack(((sub_ ∧ local_) ? 1 : 0) + 2 * dynamic_ + 4 * local_);
    if (sub_ ∧ local_) {
      master_→err( ) ≪ "ABA_CONVAR::pack():␣constraint/variable␣must␣""be␣globally␣valid." ≪
          endl;
      exit(Fatal);
    }
  }
```

154. ABA_ROW.

155. The message constructor creates the row from an **ABA_MESSAGE**.

Arguments:

glob

A pointer to the corresponding global object.

msg

The message from which the object is initialized.

```
⟨ row.cc 155 ⟩ ≡
ABA_ROW::ABA_ROW(const ABA_GLOBAL *glob, ABA_MESSAGE &msg):
  ABA_SPARVEC(glob, msg),
  sense_(glob, msg)
  {
    msg.unpack(rhs_);
  }
```

See also section 156.

156. The function *pack*( ) packs the data of the row in an **ABA_MESSAGE** object.

Arguments:

msg

The **ABA_MESSAGE** object in which the row is packed.

```
⟨ row.cc 155 ⟩ +≡
  void ABA_ROW::pack(ABA_MESSAGE &msg) const
  {
    ABA_SPARVEC::pack(msg);
    sense_.pack(msg);
    msg.pack(rhs_);
  }
```

157. ABA_COLUMN.

158. The message constructor creates the column from an **ABA_MESSAGE**.

Arguments:

glob
: A pointer to the corresponding global object.

msg
: The message from which the object is initialized.

```
⟨ column.cc 158 ⟩ ≡
ABA_COLUMN::ABA_COLUMN(const ABA_GLOBAL *glob, ABA_MESSAGE &msg):
  ABA_SPARVEC(glob, msg)
  {
    msg.unpack(obj_);
    msg.unpack(lBound_);
    msg.unpack(uBound_);
  }
```

See also section 159.

159. The function *pack*( ) packs the data of the column in an **ABA_MESSAGE** object.

Arguments:

msg
: The **ABA_MESSAGE** object in which the column is packed.

```
⟨ column.cc 158 ⟩ +≡
  void ABA_COLUMN::pack(ABA_MESSAGE &msg) const
  {
    ABA_SPARVEC::pack(msg);
    msg.pack(obj_);
    msg.pack(lBound_);
    msg.pack(uBound_);
  }
```

160. ABA_NUMCON.

161. The message constructor creates the constraint from an **ABA_MESSAGE**.

Arguments:

msg

The **ABA_MESSAGE** object from which the constraint is initialized.

```
⟨ numcon.cc 161 ⟩ ≡
ABA_NUMCON::ABA_NUMCON(const ABA_MASTER *master, ABA_MESSAGE &msg):
  ABA_CONSTRAINT(master, msg)
  {
    msg.unpack(number_);
  }
```

See also sections 162 and 163.

162. The function *pack*( ) packs the data of the constraint in an **ABA_MESSAGE** object.

Arguments:

msg

The **ABA_MESSAGE** object in which the constraint is packed.

```
⟨ numcon.cc 161 ⟩ +≡
  void ABA_NUMCON::pack(ABA_MESSAGE &msg) const
  {
    ABA_CONSTRAINT::pack(msg);
    msg.pack(number_);
  }
```

163. The function *classId*( ) returns an integer which identifies the class.

Return Value:

The class identification.

```
⟨ numcon.cc 161 ⟩ +≡
  int ABA_NUMCON::classId() const
  {
    return NUMCON_CLASSID;
  }
```

164. ABA_ROWCON.

165. The message constructor creates the constraint from an **ABA_MESSAGE**.

Arguments:

msg

The **ABA_MESSAGE** object from which the constraint is initialized.

```
⟨ rowcon.cc 165 ⟩ ≡
ABA_ROWCON::ABA_ROWCON(const ABA_MASTER *master, ABA_MESSAGE &msg):
  ABA_CONSTRAINT(master, msg),
  row_(master, msg)
  { }
```

See also sections 166 and 167.

166. The function *pack*( ) packs the data of the constraint in an **ABA_MESSAGE** object.

Arguments:

msg

The **ABA_MESSAGE** object in which the constraint is packed.

```
⟨ rowcon.cc 165 ⟩ +≡
  void ABA_ROWCON::pack(ABA_MESSAGE &msg) const
  {
    ABA_CONSTRAINT::pack(msg);
    row_.pack(msg);
  }
```

167. The function *classId*( ) returns an integer which identifies the class.

Return Value:

The class identification.

```
⟨ rowcon.cc 165 ⟩ +≡
  int ABA_ROWCON::classId( ) const
  {
    return ROWCON_CLASSID;
  }
```

168. ABA_NUMVAR.

169. The message constructor creates the variable from an **ABA_MESSAGE**.

Arguments:

msg

The **ABA_MESSAGE** object from which the variable is initialized.

```
⟨ numvar.cc 169 ⟩ ≡
ABA_NUMVAR::ABA_NUMVAR(const ABA_MASTER *master, ABA_MESSAGE &msg):
  ABA_VARIABLE(master, msg)
  {
    msg.unpack(number_);
  }
```

See also sections 170 and 171.

170. The function *pack*( ) packs the data of the variable in an **ABA_MESSAGE** object.

Arguments:

msg

The **ABA_MESSAGE** object in which the variable is packed.

```
⟨ numvar.cc 169 ⟩ +≡
  void ABA_NUMVAR::pack(ABA_MESSAGE &msg) const
  {
    ABA_VARIABLE::pack(msg);
    msg.pack(number_);
  }
```

171. The function *classId*( ) returns an integer which identifies the class.

Return Value:

The class identification.

```
⟨ numvar.cc 169 ⟩ +≡
  int ABA_NUMVAR::classId( ) const
  {
    return NUMVAR_CLASSID;
  }
```

172. ABA_SROWCON.

173. The message constructor creates the constraint from an **ABA_MESSAGE**.

Arguments:

msg
The **ABA_MESSAGE** object from which the constraint is initialized.

```
⟨ srowcon.cc 173 ⟩ ≡
ABA_SROWCON::ABA_SROWCON(const ABA_MASTER *master, ABA_MESSAGE &msg):
  ABA_ROWCON(master, msg)
  { }
```

See also sections 174 and 175.

174. The function *pack*( ) packs the data of the constraint in an **ABA_MESSAGE** object.
This function needs not to be implemented here because it just calls the *pack*( ) function of the base class. It is provided for the convenience of the user only.

Arguments:

msg
The **ABA_MESSAGE** object in which the constraint is packed.

```
⟨ srowcon.cc 173 ⟩ +≡
  void ABA_SROWCON::pack(ABA_MESSAGE &msg) const
  {
    ABA_ROWCON::pack(msg);
  }
```

175. The function *classId*( ) returns an integer which identifies the class.

Return Value:

The class identification.

```
⟨ srowcon.cc 173 ⟩ +≡
  int ABA_SROWCON::classId( ) const
  {
    return SROWCON_CLASSID;
  }
```

176. ABA_COLVAR.

177. The message constructor creates the variable from an **ABA_MESSAGE**.

Arguments:

msg

The **ABA_MESSAGE** object from which the variable is initialized.

```
⟨ colvar.cc 177 ⟩ ≡
ABA_COLVAR::ABA_COLVAR(const ABA_MASTER *master, ABA_MESSAGE &msg):
  ABA_VARIABLE(master, msg),
  column_(master, msg)
  { }
```

See also sections 178 and 179.

178. The function *pack*( ) packs the data of the variable in an **ABA_MESSAGE** object.

Arguments:

msg

The **ABA_MESSAGE** object in which the variable is packed.

```
⟨ colvar.cc 177 ⟩ +≡
  void ABA_COLVAR::pack(ABA_MESSAGE &msg) const
  {
    ABA_VARIABLE::pack(msg);
    column_.pack(msg);
  }
```

179. The function *classId*( ) returns an integer which identifies the class.

Return Value:

The class identification.

```
⟨ colvar.cc 177 ⟩ +≡
  int ABA_COLVAR::classId( ) const
  {
    return COLVAR_CLASSID;
  }
```

180. ABA_ACTIVE.

181. The message constructor creates the object from an **ABA_MESSAGE**. The **ABA_POOLSLOTREF** references to the constraints/variables which are locally available are initialized. The indexes of the missing constraints/variables are collected in an **ABA_BUFFER**.

To complete the initialization of the object the function *unpackNeeded*( ) has to be called afterwards.

Arguments:

msg
The **ABA_MESSAGE** object from which the object is constructed.

idBuffer
Returns a pointer to an **ABA_BUFFER** containing the IDs of all referenced constraints/variables. $*idBuffer$ is allocated in this function and has to be freed by the caller.

needed
Returns a pointer to an **ABA_BUFFER** containing the indexes of the constraints/variables not locally available. $*needed$ is allocated in this function and has to be freed by the caller.

```
⟨ active.inc 181 ⟩ ≡
template⟨class BaseType, class CoType⟩ ABA_ACTIVE⟨BaseType, CoType⟩::ABA_ACTIVE(const
      ABA_MASTER *master, ABA_MESSAGE &msg, ABA_BUFFER⟨ABA_ID⟩
      **idBuffer, ABA_BUFFER⟨int⟩ **needed):
  master_(master),
  n_(msg.unpackInt( )),
  active_(master, msg.unpackInt( )),
  redundantAge_(master, msg)
  {
    *idBuffer = new ABA_BUFFER⟨ABA_ID⟩ (master, n_);
    *needed = new ABA_BUFFER⟨int⟩ (master, n_);
    for (int i = 0; i < n_; i++) {
      ABA_ID id(msg);
      (*idBuffer)→push(id);
      if (id.isInitialized( )) {
        ABA_POOL⟨BaseType, CoType⟩ *pool = (ABA_POOL⟨BaseType, CoType⟩ *)
            master_→parmaster( )→getPool(id.index( ));
        ABA_POOLSLOT⟨BaseType, CoType⟩ *ps = pool→findSlot(id);
        if (ps) active_[i] = new ABA_POOLSLOTREF⟨BaseType, CoType⟩ (ps);
        else {
          (*needed)→push(i);
          active_[i] = 0;      /* will be initialized in unpackNeeded() */
        }
      }
      else      /* removed conVar */
        active_[i] = 0;
    }
  }
```

See also sections 182, 183, and 184.

182. The function *unpackNeeded*( ) unpacks the constraints/variables which were packed by the function *packNeeded*( ). The corresponding entries of the *active_* array are initialized.

Arguments:

msg

The **ABA_MESSAGE** object from which the data is unpacked.

idBuffer

A **ABA_BUFFER** with the IDs of all referenced constraints/variables.

needed

A **ABA_BUFFER** with the indexes of the constraints/variables to be unpacked.

```
⟨ active.inc 181 ⟩ +≡
  template⟨class BaseType, class CoType⟩ void ABA_ACTIVE⟨BaseType,
        CoType⟩::unpackNeeded(ABA_MESSAGE &msg, const ABA_BUFFER⟨ABA_ID⟩
        &idBuffer, const ABA_BUFFER⟨int⟩ &needed){ for (int i = 0; i < needed.number( ); i++)
        { int classId;
    msg.unpack(classId); BaseType * cv = ( BaseType * ) master_→unpackConVar(msg, classId);
    const ABA_ID &id = idBuffer[needed[i]];
    if (debug(DEBUG_MESSAGE_CONVAR)) {
      master_→out( ) ≪ "DEBUG_MESSAGE_CONVAR:␣Constraint/Variable␣" ≪ id ≪ "␣(classId=" ≪
          classId ≪ ")␣received." ≪ endl;
    }
    ABA_POOL⟨BaseType, CoType⟩ *pool = (ABA_POOL⟨BaseType, CoType⟩ *)
        master_→parmaster( )→getPool(id.index( ));
    ABA_POOLSLOT⟨BaseType, CoType⟩ *ps = pool→insert(cv);
    if (ps ≡ 0) {
      master_→err( ) ≪ "ABA_ACTIVE::ABA_ACTIVE():␣no␣room␣to␣in\
          sert␣constraint""␣into␣pool." ≪ endl;
      exit(Fatal);
    }
    ps→setIdentification(id);
    active_[needed[i]] = newABA_POOLSLOTREF⟨BaseType, CoType⟩ (ps); } }
```

183. The function *pack*( ) packs the size information of the object and the IDs of the referenced constraints/variables in an **ABA_MESSAGE** object.

Arguments:

msg

The **ABA_MESSAGE** object in which the IDs are packed.

```
⟨active.inc 181⟩ +≡
  template⟨class BaseType, class CoType⟩ void ABA_ACTIVE⟨BaseType,
          CoType⟩::pack(ABA_MESSAGE &msg) const
  {
    msg.pack(n_);
    msg.pack(active_.size());
    redundantAge_.pack(msg, n_);
    ABA_ID uninitializedId;
    for (int i = 0; i < n_; i++) {
      if (active_[i]→conVar( ) ≡ 0)      /* removed conVar */
        uninitializedId.pack(msg);
      else {
        ABA_POOLSLOT⟨BaseType, CoType⟩ *ps = active_[i]→slot( );
        if (¬ps→getIdentification( ).isInitialized( )) ps→setNewIdentification( );
        ps→getIdentification( ).pack(msg);
      }
    }
  }
```

184. The function *packNeeded*( ) packs the constraints/variables whose indexes are stored in an **ABA_BUFFER**.

Arguments:

msg

The **ABA_MESSAGE** object in which the data is packed.

needed

The indexes of the constraints/variables which should be packed.

```
⟨active.inc 181⟩ +≡
  template⟨class BaseType,
          class CoType⟩ void ABA_ACTIVE⟨BaseType, CoType⟩::packNeeded(ABA_MESSAGE
          &msg, const ABA_BUFFER⟨int⟩ &needed) const
  {
    for (int i = 0; i < needed.number( ); i++) {
      BaseType * cv = active_[needed[i]]→conVar( );
      if (debug(DEBUG_MESSAGE_CONVAR)) {
        master_→out( ) ≪ "DEBUG_MESSAGE_CONVAR:␣sending␣Constraint/Variable␣" ≪
            active_[needed[i]]→slot( )→getIdentification( ) ≪ "␣(classId=" ≪ cv→classId( ) ≪ ")..." ≪
            endl;
      }
      msg.pack(cv→classId( ));
      cv→pack(msg);
    }
  }
```

185. ABA_OPENSUB.

186. The function *terminate*( ) terminates the function *select*( ).

```
⟨ opensub.cc 186 ⟩ ≡
  void ABA_OPENSUB::terminate()
  {
    terminate_ = true;
    while (¬hasTerminated_) cond_.signal();
  }
```

See also sections 187 and 188.

187. The function *getSubproblemWithBound*( ) extracts the best subproblem of the list of open subproblems if its bound is at least as good as the given value.

Arguments:
  bound
    The requested dual bound

Return Value:
  A pointer to such a subproblem or 0.

```
⟨ opensub.cc 186 ⟩ +≡
  ABA_SUB *ABA_OPENSUB::getSubproblemWithBound(double bound){ mutex_.acquire();
        ABA_DLISTITEM < ABA_SUB * > *item;
      ABA_SUB *s;
      forAllDListElem(list_, item, s)
      {
        if (s→status( ) ≡ ABA_SUB::Dormant) {
          s→newDormantRound( );
          if (s→nDormantRounds( ) < master_→minDormantRounds( )) continue;
        }     /* is the bound of s good enough? */
        if (master_→optSense( )→max( )) {
          if (s→dualBound( ) ≥ bound) break;
        }
        else {
          if (s→dualBound( ) ≤ bound) break;
        }
      }
      if (item ≡ 0) {
        mutex_.release( );
        return 0;
      }     /* subproblem found */
      assert(s ≡ item→elem( ));
      list_.remove(item);
      updateDualBound( );
      mutex_.release( );
      master_→parmaster( )→newOpenSubCount(n_, dualBound_);
      return s; }
```

188. The function *select*( ) selects a subproblem according to the strategy in *master* and removes it from the list of open subproblems.

The function *select*( ) scans the list of open subproblems, and selects the subproblem with highest priority from the set of open subproblems. Dormant subproblems are ignored if possible.

Return Value:

The selected subproblem. If the set of open subproblems is empty, 0 is returned.

```
⟨ opensub.cc 186 ⟩ +≡
  ABA_SUB *ABA_OPENSUB::select(){ mutex_.acquire(); while (¬terminate_) {
      /* perform load balancing if needed */
    mutex_.release();
    for (int i = 0; i < 3; i++)
      if (master_→parmaster()→balance()) break;
    mutex_.acquire();      /* select the next subproblem to optimize */
    ABA_DLISTITEM < ABA_SUB * > *minItem = list_.first(); ABA_DLISTITEM < ABA_SUB
        * > *item;
    ABA_SUB *s;
    forAllDListElem(list_, item, s)
    {
      if (s→status() ≡ ABA_SUB::Dormant) {
        s→newDormantRound();
        if (s→nDormantRounds() < master_→minDormantRounds()) continue;
      }
      if (master_→enumerationStrategy(s, minItem→elem()) > 0) minItem = item;
    }
    if (minItem ≠ 0) {      /* subproblem found */
      ABA_SUB *min = minItem→elem();
      list_.remove(minItem);
      updateDualBound();
      mutex_.release();
      master_→parmaster()→newOpenSubCount(n_, dualBound_);
      return min;
    }     /* host zero should poll for termination */
    if (master_→parmaster()→isHostZero()) {
      ABA_NOTIFICATION msg(master_, 0);
      msg.pack(ABA_NOTIFYSERVER::TriggerTerminationCheckTag);
      msg.send();
      ACE_Time_Value abstime(0, 400000);
      abstime += ACE_OS::gettimeofday();
      master_→parmaster()→startIdleTime();
      cond_.wait(&abstime);      /* wait at most 0.4 s */
      master_→parmaster()→stopIdleTime();
    }
    else {      /* the other hosts should wait until signaled */
      master_→parmaster()→startIdleTime();
      cond_.wait();
      master_→parmaster()→stopIdleTime();
    }
    } mutex_.release();
    hasTerminated_ = true;      /* terminate ABA_OPENSUB::terminate() */
    return 0; }
```

189. ABA_SPARVEC.

190. The message constructor creates the sparse vector from an **ABA_MESSAGE**.

Arguments:

glob

A pointer to the corresponding global object.

msg

The message from which the object is initialized.

```
⟨ sparvec.cc 190 ⟩ ≡
ABA_SPARVEC::ABA_SPARVEC(const ABA_GLOBAL *glob, ABA_MESSAGE &msg):
  glob_(glob)
  {
    msg.unpack(size_);
    msg.unpack(nnz_);
    msg.unpack(reallocFac_);
    if (size_) {
      support_ = new int [size_];
      coeff_ = new double [size_];
      msg.unpack(support_, nnz_);
      msg.unpack(coeff_, nnz_);
    }
    else {
      support_ = 0;
      coeff_ = 0;
    }
  }
```

See also section 191.

191. The function *pack*( ) packs the data of the sparse vector in an **ABA_MESSAGE** object.

Arguments:

msg

The **ABA_MESSAGE** object in which the sparse vector is packed.

```
⟨ sparvec.cc 190 ⟩ +≡
  void ABA_SPARVEC::pack(ABA_MESSAGE &msg) const
  {
    msg.pack(size_);
    msg.pack(nnz_);
    msg.pack(reallocFac_);
    if (size_) {
      msg.pack(support_, nnz_);
      msg.pack(coeff_, nnz_);
    }
  }
```

192. ABA_STRING.

193. The message constructor creates the **ABA_STRING** from an **ABA_MESSAGE**.

Arguments:

msg
The **ABA_MESSAGE** object from which the string is initialized.

```
⟨ string.cc 193 ⟩ ≡
ABA_STRING::ABA_STRING(const ABA_GLOBAL *glob, ABA_MESSAGE &msg):
  glob_(glob),
  string_(0)
  {
    unpack(msg);
  }
```

See also sections 194 and 195.

194. The function *unpack*( ) unpacks the string from an **ABA_MESSAGE** object.

Arguments:

msg
The **ABA_MESSAGE** object from which the string is unpacked.

```
⟨ string.cc 193 ⟩ +≡
  void ABA_STRING::unpack(ABA_MESSAGE &msg)
  {
    int size;
    msg.unpack(size);
    delete []string_;
    string_ = new char [size + 1];
    msg.unpack(string_, size);
    string_[size] = '\0';
  }
```

195. The function *pack*( ) packs the string in an **ABA_MESSAGE** object.

Arguments:

msg
The **ABA_MESSAGE** object in which the string is packed.

```
⟨ string.cc 193 ⟩ +≡
  void ABA_STRING::pack(ABA_MESSAGE &msg) const
  {
    int size = ::strlen(string_);
    msg.pack(size);
    msg.pack(string_, size);
  }
```

196. ABA_ARRAY.

197. The message constructor creates the **ABA_ARRAY** from an **ABA_MESSAGE**.

Arguments:

glob
A pointer to the corresponding global object.

msg
The message from which the object is initialized.

```
⟨ array.inc 197 ⟩ ≡
template⟨class Type⟩ ABA_ARRAY⟨Type⟩::ABA_ARRAY(const ABA_GLOBAL
    *glob, ABA_MESSAGE &msg):
  glob_(glob),
  n_(0),
  a_(0)
  {
    glob_→err() ≪ "ABA_ARRAY::ABA_ARRAY()␣:␣An␣ABA_ARRAY␣o\
        f␣some␣type␣coudn't""␣be␣received.␣You␣have␣to␣implement␣a␣t\
        emplate""␣specialization␣of␣the␣message␣constructor␣for""␣that␣type!" ≪ endl;
    exit(Fatal);
  }
```

See also sections 198 and 199.

198. The function *pack*( ) packs all elements of the array in an **ABA_MESSAGE** object.

Arguments:

msg
The **ABA_MESSAGE** object in which the array is packed.

```
⟨ array.inc 197 ⟩ +≡
  template⟨class Type⟩ void ABA_ARRAY⟨Type⟩::pack(ABA_MESSAGE &msg) const
  {
    pack(msg, n_);
  }
```

199. This version of the function *pack*( ) packs a limited number of elements of the array in an **ABA_MESSAGE** object, only.

Arguments:

msg
The **ABA_MESSAGE** object in which the array is packed.

nPacked
The number of elements to be packed.

```
⟨ array.inc 197 ⟩ +≡
  template⟨class Type⟩ void ABA_ARRAY⟨Type⟩::pack(ABA_MESSAGE &msg, int nPacked)
        const
  {
    glob_→err() ≪ "ABA_ARRAY::pack()␣:␣An␣ABA_ARRAY␣of␣som\
        e␣type␣coudn't""␣be␣sent.␣You␣first␣have␣to␣implement␣a\
        ␣template""␣specialization␣of␣the␣pack()␣template␣function""␣for␣that␣type!" ≪
        endl;
    exit(Fatal);
  }
```

200. ABA_BUFFER.

201. The message constructor creates the **ABA_BUFFER** from an **ABA_MESSAGE**.

Arguments:

glob
A pointer to the corresponding global object.

msg
The message from which the object is initialized.

```
⟨ buffer.inc 201 ⟩ ≡
template⟨class Type⟩ ABA_BUFFER⟨Type⟩::ABA_BUFFER(const ABA_GLOBAL
    *glob, ABA_MESSAGE &msg):
  glob_(glob)
  {
    glob_→err() ≪ "ABA_BUFFER::ABA_BUFFER()␣:␣A␣ABA_BUFFER␣of␣some␣type␣coudn't\
        t""␣be␣received.␣You␣have␣to␣implement␣a␣template""␣specialization␣of␣\
        the␣message␣constructor␣for""␣␣that␣type!" ≪ endl;
    exit(Fatal);
  }
```

See also section 202.

202. The function *pack*( ) packs the data of the buffer in an **ABA_MESSAGE** object.

Arguments:

msg
The **ABA_MESSAGE** object in which the buffer is packed.

```
⟨ buffer.inc 201 ⟩ +≡
  template⟨class Type⟩ void ABA_BUFFER⟨Type⟩::pack(ABA_MESSAGE &msg) const
  {
    glob_→err() ≪ "ABA_BUFFER::pack()␣:␣A␣ABA_BUFFER␣of␣so\
        me␣type␣coudn't""␣be␣sent.␣You␣first␣have␣to␣implement␣a\
        ␣template""␣specialization␣of␣the␣pack()␣template␣function""␣for␣that␣type!" ≪
        endl;
    exit(Fatal);
  }
```

203. ABA_HASH.

204. This is a hash function for elements of the class **ABA_ID**. A different random number associated with each processor is used to shuffle identical sequence numbers to different locations in the hash table.

```
⟨ hash.inc 204 ⟩ ≡
  template⟨class KeyType, class ItemType⟩ int ABA_HASH < KeyType, ItemType > ::hf(const
          ABA_ID &id)
  {
    const int rand[64] = {#08fa735c, #465969d0, #66a657f4, #144d2cf9, #32b20675, #7d86036c,
        #3bcd2f61, #30421197, #272d9013, #1d3bf099, #1bd38ed1, #57abc10e, #7e62fbf6, #0b9bf7ad,
        #15bd99d9, #451a0198, #73b3a879, #325eeb8a, #1dbb0b7c, #5bec0be6, #2e78432e, #2e2ceea6,
        #55177a1a, #7b31a98f, #54d04dd5, #547bd0d0, #1d12c33a, #16fb478f, #687e3120, #4a047b2e,
        #649e29fb, #1c36b5ae, #3a9e8db8, #6488c827, #5b6315fa, #60b4e7c1, #5c116177, #336ead28,
        #7dcdd34c, #41b4bb6e, #3f7aeaa3, #687cf590, #19469807, #56a508f0, #179ed4c4, #06e73a00,
        #007da2a3, #41e5ac24, #0585b479, #5b1cf529, #285b5b9a, #3bbaea37, #7c84f882, #081c97ba,
        #6df23bc6, #1f655ecb, #291ac2ac, #7598ef40, #5b8235b8, #25ccaa59, #65a52132, #2cf89028,
        #1d05cf45, #32e86c2b};
    return (rand[id.proc( ) & 63] ⊕ id.sequence( )) % size_;
  }
```

205. REFERENCES.

[ACE] D. C. Schmidt, The ADAPTIVE Communication Environment (ACE), *Washington University*, http://www.cs.wustl.edu/~schmidt/ACE.html.

[Ref22] ABACUS 2.2, User's Guide and Reference Manual, HTML version, Universität zu Köln, Universität Heidelberg, 1998, http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus/html/manual.html.

[Dist22] ABACUS 2.2, Software Distribution, Universität zu Köln, Universität Heidelberg, 1998, http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus/distribution.html.

206. INDEX AND SECTION NAMES.