Angewandte Mathematik und Informatik
Universität zu Köln

Report No. 99.358

Parallel ABACUS – Introduction and Tutorial

by
Max Böhm

1999

partially supported by DFG-Projekt Ju 204/4-2, Re 776/5-3

Universität zu Köln
Institut für Informatik
Pohligstr. 1
D-50969 Köln

Parallel ABACUS – Introduction and Tutorial

Max Böhm

June 16, 1999

1 Introduction

This document describes the current state of the parallel version of ABACUS (version 2.3 beta) [4]. It should serve as an introduction to the design of the system as well as a tutorial for the user. The tutorial section describes how to extend and run the ABACUS example application in a parallel environment. For further information on the sequential ABACUS see [1, 2, 9]

The *Parallel ABACUS* performs a tree decomposition of the Branch&Cut tree. Each host of the workstation cluster runs the sequential ABACUS code on a different node of the tree. The system is designed to run fully distributed, i.e. there is no master controlling the whole system. Each host keeps state information of the system and decides by itself what to do next. This eliminates the bottleneck which typically occurs in master/slave architectures.

The parallel version of ABACUS can be used mostly like a blackbox. A user needs to write a few methods for serialization of his objects and replace the sequential ABACUS library by the parallel ABACUS library.

The parallel ABACUS was designed to be easily extendable. It currently does not contain features like parallel separation and distributed variable and constraint pools. These should be added in a future version.

The complete source code is documented in [3].

2 Architecture

The architecture of the parallel ABACUS is shown in figure 1. The system is designed work be multithreaded. This means that different components can run in parallel in different threads. If one thread blocks for a communication this does not affect the other threads. Standard communication libraries like MPI or PVM often are not threadsafe. Therfore we used the *Adaptive Communication Environment (ACE)* library [7] for communication and thread management. ACE is an object oriented C++ library which is threadsafe, portable and open source. Among the supported platforms are most Unix versions, Linux and Windows NT. ACE also supports design patterns like *Active Objects* and others, see [5, 8].

The parallel ABACUS system consists of the following components.

2.1 ABACUS Kernel

The ABACUS kernel is running the Branch&Cut algorithm on each host. First a new subproblem is selected for optimization. The user can choose between the strategies *Local Best First*, *Global Best First* and *Hybrid*. *Local Best First* selects the subproblem with the best dual bound of the local set of open subproblems. *Global Best First* requests some (possibly remote) host for the subproblem with the global best dual bound like in a *work stealing* strategy. The *Hybrid*

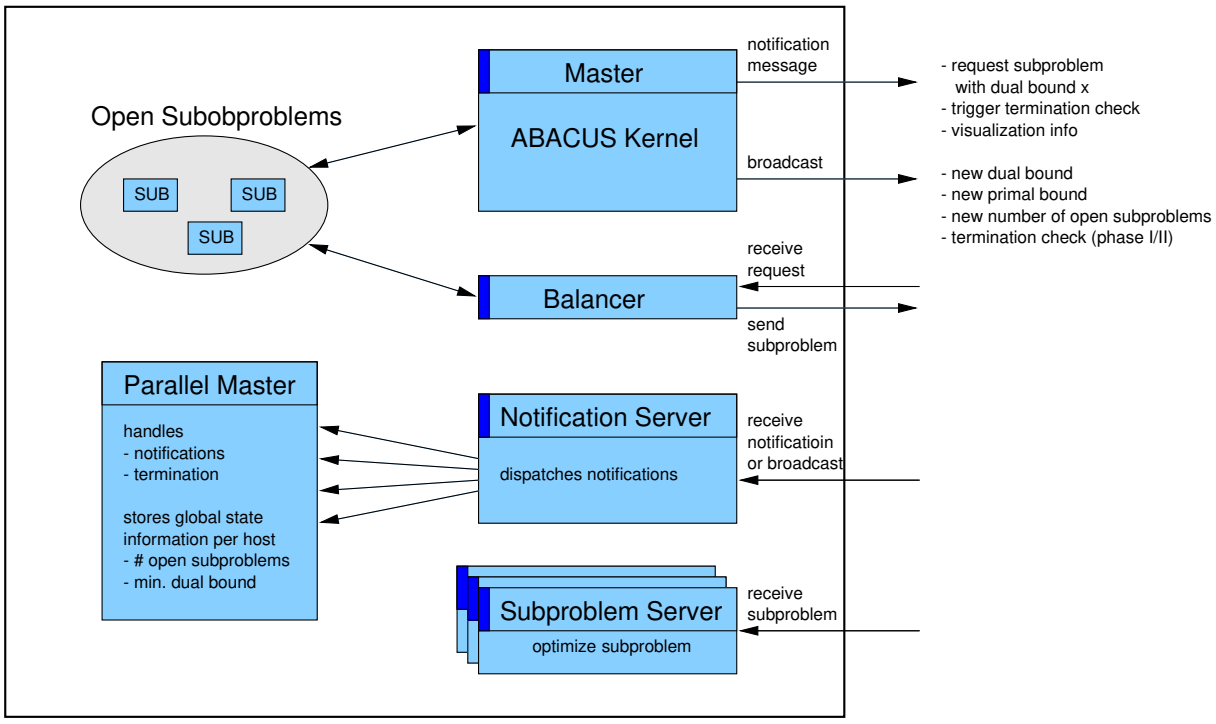

Figure 1: Architecture of a host of the parallel system. Components marked at the left side run within their own thread.

strategy selects the best subproblem of the local set if the difference of its dual bound and the best known dual bound compared to the difference of the best known primal bound and the best known dual bound is within some ratio given in percent.

While the cutting plane algorithm is running new dual bounds are broadcast asynchronously as soon as they become known. When the system decides to branch on the subproblem the two children are input in the local set of open subproblems and the information on the number of open subproblems in the local set and the best local dual bound are broadcast.

2.2 Balancer

The balancer thread is currently very simple. It handles incoming requests for subproblems of some given lower bound. If such a subproblem is locally available it is sent to the requesting host. This is done in parallel to the optimization in process at that host. In a future version the balancer should also perform some load balancing algorithm continuously in the background.

2.3 Notification Server

The notification server runs in a separate thread and listens for notification messages or notification broadcasts from other processors. The messages are then dispatched to the various handlers most of which are located in the class `ABA_PARMASTER`.

2.4 Parallel Master

The parallel master contains handlers for several notification messages and keeps global state information of the system, namely the number of open subproblems at each host and the best

local dual bound of each host. The parallel master is also responsible for initialization, parameter handling, and termination detection.

2.5 Subproblem Solver

The subproblem solver is a multithreaded server for optimizing subproblems in parallel on a single SMP host with shared memory. This component is currently not in use but reserved for a future version of the parallel ABACUS.

3 Unique Identification of Objects

Each object gets a globally unique identification when beeing created (implemented by the class `ABA_ID`). This identification is composed of a host number where the object was first created and a unique sequence number. The identifications of all objects local to a host are stored in a local hash table (implemented by the class `ABA_IDMAP`). When an object is sent over the network, it can be easily determined, if a referenced object is already residing in the memory of the destination host or not. This can be done by sending the IDs of the referenced objects instead of pointers and looking up these IDs in the hash table at the destination host. In case of a hit, a copy of the referenced object is available at the destination system and a pointer to it is returned. Otherwise the referenced object is missing and has to be transferred over the network. Using this strategy the number of objects (variables and constraints) which have to be sent over the network is minimized.

4 Serialization of Objects

The user has to implement virtual member functions used by the framework for sending/receiveing objects of the problem specific subclasses of `ABA_VARIABLE`, `ABA_CONSTRAINT` and `ABA_SUB`. The virtual member function `pack(ABA_MESSAGE&)` is called by the framework when the data of an object should be packed in a message which is then sent to another host. The class `ABA_MESSAGE` provides a number of member functions which can be used to pack basic datatypes and arrays, see the header file `message.h` for details. At the receiving host the system creates an object from a message by using the *message constructor* `MYCLASS::MYCLASS(ABA_MESSAGE&)`. Additionally the user has to implement the virtual function `classId()` in problem specific subclasses of `ABA_CONSTRAINT`, `ABA_VARIABLE` and `ABA_BRANCHRULE` which returns a unique integer for that class. This is used for the simulation of runtime type information. In the problem specific subclass of `ABA_MASTER` the virtual functions `unpackConVar(ABA_MESSAGE&, int classId)` and `unpackSub(ABA_MESSAGE&)` have to be implemented. It is recommended to enclose all parallel extensions by `#ifdef ABACUS_PARALLEL ... #endif`. The sections below illustrate which code has to be added to the simple TSP solver example of the ABACUS distribution [10].

4.1 Parallel extensions to `edge.h` and `edge.cc`

```
#define EDGE_CLASSID 0

class EDGE : public ABA_VARIABLE {
  public:
#ifdef ABACUS_PARALLEL
    EDGE(const ABA_MASTER *master, ABA_MESSAGE &msg);
```

```
    virtual void pack(ABA_MESSAGE &msg) const;
    virtual int classId() const { return EDGE_CLASSID; }
#endif
};

#ifdef ABACUS_PARALLEL

// The message constructor.
EDGE::EDGE(const ABA_MASTER *master, ABA_MESSAGE &msg)
  :
  ABA_VARIABLE(master, msg)
{
  msg.unpack(tail_);
  msg.unpack(head_);
}

// The function pack().
void EDGE::pack(ABA_MESSAGE &msg) const
{
  ABA_VARIABLE::pack(msg);
  msg.pack(tail_);
  msg.pack(head_);
}

#endif
```

4.2 Parallel extensions to `degree.h` and `degree.cc`

```
#define DEGREE_CLASSID 1

class DEGREE : public ABA_CONSTRAINT {
  public:
#ifdef ABACUS_PARALLEL
    DEGREE(const ABA_MASTER *master, ABA_MESSAGE &msg);
    virtual void pack(ABA_MESSAGE &msg) const;
    virtual int classId() const { return DEGREE_CLASSID; }
#endif
};

#ifdef ABACUS_PARALLEL

// The message constructor.
DEGREE::DEGREE(const ABA_MASTER *master, ABA_MESSAGE &msg)
  :
  ABA_CONSTRAINT(master, msg)
{
  msg.unpack(node_);
}
```

```
// The function pack().
void DEGREE::pack(ABA_MESSAGE &msg) const
{
  ABA_CONSTRAINT::pack(msg);
  msg.pack(node_);
}

#endif
```

4.3 Parallel extensions to `subtour.h` and `subtour.cc`

```
#define SUBTOUR_CLASSID 2

class SUBTOUR : public ABA_CONSTRAINT {
  public:
#ifdef ABACUS_PARALLEL
    SUBTOUR(const ABA_MASTER *master, ABA_MESSAGE &msg);
    virtual void pack(ABA_MESSAGE &msg) const;
    virtual int classId() const { return SUBTOUR_CLASSID; }
#endif
};

#ifdef ABACUS_PARALLEL

// The message constructor.
SUBTOUR::SUBTOUR(const ABA_MASTER *master, ABA_MESSAGE &msg)
  :
  ABA_CONSTRAINT(master, msg),
  nodes_(master, msg),
  marked_(0)
{ }

// The function |pack()|.
void SUBTOUR::pack(ABA_MESSAGE &msg) const
{
  ABA_CONSTRAINT::pack(msg);
  nodes_.pack(msg);
}

#endif
```

4.4 Parallel extensions to `tspmaster.h` and `tspmaster.cc`

```
class TSPMASTER : public ABA_MASTER {
  public:
#ifdef ABACUS_PARALLEL
    ABA_CONVAR* unpackConVar(ABA_MESSAGE &msg, int classId) const;
    ABA_SUB* unpackSub(ABA_MESSAGE &msg);
```

```
#endif
};

#ifdef ABACUS_PARALLEL

#include "abacus/parmaster.h"
#include "subtour.h"

// The virtual function unpackConVar() constructs and unpacks an
// object of some subclass of ABA_CONVAR from a Message. The function is
// called when an object of some subclass of ABA_CONSTRAINT
// or ABA_VARIABLE has to be recreated at the receiving processor.
// The unpackConVar() function of the base class has to be called,
// if the class identification does not belong to a user defined subclass.
//
// Return:
//   A Pointer to a newly constructed object of the subclass of
//   ABA_CONVAR specified by classId.
//
// Arguments:
//   msg           The message from which the ABA_CONVAR is unpacked.
//   classId       The class identification of the subclass of ABA_CONVAR
//                 which should be unpacked.
//
ABA_CONVAR* TSPMASTER::unpackConVar(ABA_MESSAGE &msg, int classId) const
{
  switch (classId) {
    case EDGE_CLASSID:
      return new EDGE(this, msg);
    case DEGREE_CLASSID:
      return new DEGREE(this, msg);
    case SUBTOUR_CLASSID:
      return new SUBTOUR(this, msg);
  }
  return ABA_MASTER::unpackConVar(msg, classId);
}

// The virtual function unpackSub() constructs and unpacks an object
// of the user defined subclass of ABA_SUB from a message.
//
// Return:
//   A Pointer to a newly constructed object of the user defined subclass of
//   ABA_SUB.
//
// Arguments:
//   msg           The message from which the ABA_SUB is unpacked.
//
ABA_SUB* TSPMASTER::unpackSub(ABA_MESSAGE &msg)
```

```
{
  return new TSPSUB(this, msg);
}

#endif
```

4.5 Parallel extensions to `tspmaster.h` and `tspmaster.cc`

```
class TSPSUB : public ABA_SUB {
  public:
#ifdef ABACUS_PARALLEL
    TSPSUB(ABA_MASTER *master, ABA_MESSAGE &msg);
#endif

#ifdef ABACUS_PARALLEL

// The message constructor creates the subproblem from an ABA_MESSAGE by
// calling the message constructor of the base class.
//
// Arguments
//   msg           The ABA_MESSAGE object from which the subproblem is
//                 initialized.
//
TSPSUB::TSPSUB(ABA_MASTER *master, ABA_MESSAGE &msg)
:
  ABA_SUB(master, msg)
{ }

#endif
```

5 Parallel Execution

The application has to be compiled with the preprocessor flag `ABACUS_PARALLEL` beeing defined. In addition to the parallel ABACUS library the ACE library [7] has to be linked to the executable.

In the file `.abacus` a number of parameters specific to the parallel version exist. They are listed below

```
#
# THE NUMBER OF HOSTS
#

ParallelHostCount 3

#
# THE HOSTNAMES
#
# These parameters have the form "ParallelHostname_%d" where
# %d = 0 ... ParallelHostCount-1
```

```
#
# The paralell ABACUS executable has to be started on each host.
# This can be done by using the "start" script which parses this
# file to determine the hostnames.
#
# The hostnames must be different from each other!
#

ParallelHostname_0 rubens
ParallelHostname_1 frueh
ParallelHostname_2 sion

#
# BEST FIRST SEARCH TOLERANCE
#
# this parameter controls the strategy for selecting the next subproblem to
# be processed by the Branch&Bound algorithm. Candidates for the next
# subproblem to be selected are the subproblem with the best bound of the
# local list of open subproblems and the subproblem with global best bound.
#
# The parameter ParallelBestFirstTolerance specifies the accepted
# tolerance of the dual bound of the best subproblem in the local list
# with respect to the intervall [global best dual bound, best primal bound]
# in percent.
#
# ParallelBestFirstTolerance =   0     means GLOBAL BEST FIRST
# ParallelBestFirstTolerance = 100     means LOCAL BEST FIRST
#
# values inbetween can be used to reduce communication while nearly
# maintaining a global best first order.
#

ParallelBestFirstTolerance 0

#
# THE DEBUG LEVEL
#
# the debug level can be any combination (sum) of the following bitmasks
#
#   DEBUG_MESSAGE_CONVAR      1
#   DEBUG_MESSAGE_SUB         2
#   DEBUG_NOTIFICATION        4
#   DEBUG_BALANCER            8
#   DEBUG_TERMINATION         16
#   DEBUG_SOCKET              32
#   DEBUG_SEPARATE            64
#   DEBUG_SUBSERVER           128
```

```
ParallelDebugLevel 0

#
# THE CONNECTION TIMEOUT
#
# timeout for connecting to other hosts in seconds
#

ParallelConnectTimeout 30

#
# THE PORT NUMBERS
#
# unix port numbers to be used for socket communication
#

ParallelNotifyPort 23463
ParallelSubproblemPort 23464
ParallelBalancerPort 23465
```

At least the number of hosts and their names have to be defined. The executable of the parallel application has to be started on each host listed in the parameter file. This can be done by using the `start` script which is contained in the distribution. The script assumes that the `rsh` command works without beeing asked for a password. To start the application simply prepend the string `start` to the command line, e.g.

```
start tsp bier127.tsp
```

All output appears in the same window. Alternatively the `start` script supports to open a separate window for each host as well as starting the GNU debugger `gdb` on each host.

6 Online Tree Visialization

It is possible to generate an online tree visualization of the exploration of the Branch&Bound tree by using the VBC tool [6]. The ABACUS output can be connected to the VBC tool via a "named pipe" which can be created by the unix command `mknod xxxx p`. In `.abacus` set `VbcLog=Pipe` and the new parameter `VbcPipeName=xxxx`. Then the VBC tool can be started with standard input connected to the pipe by the command `startVbcTool <xxxx &`. Active subproblems are painted red, open subproblems are painted blue. After a subproblem is processed its color changes to a color identifying the host, on which the subproblem was processed.

7 Sample Tree Visualization Outputs

The following figures show the Branch&Bound trees for the sequential ABACUS and the parallel ABACUS with different values of the Parameter `ParallelBestFirstTolerance` for the problem instance `bier127.tsp`.

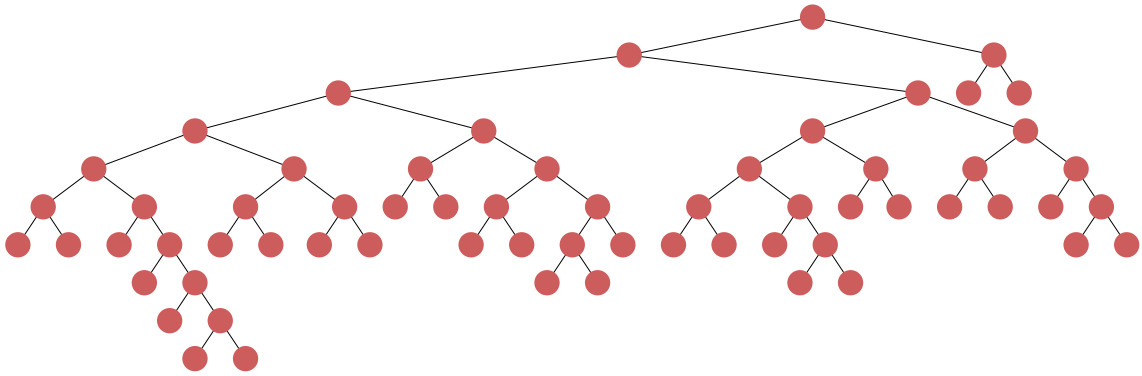

Figure 2: Branch&Bound tree generated by the sequential ABACUS, 63 nodes.

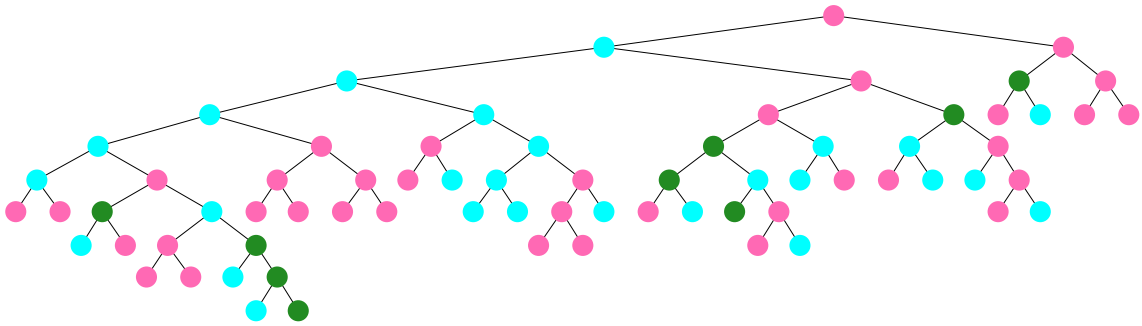

Figure 3: Parallel ABACUS, Global Best First Strategy, 71 nodes.

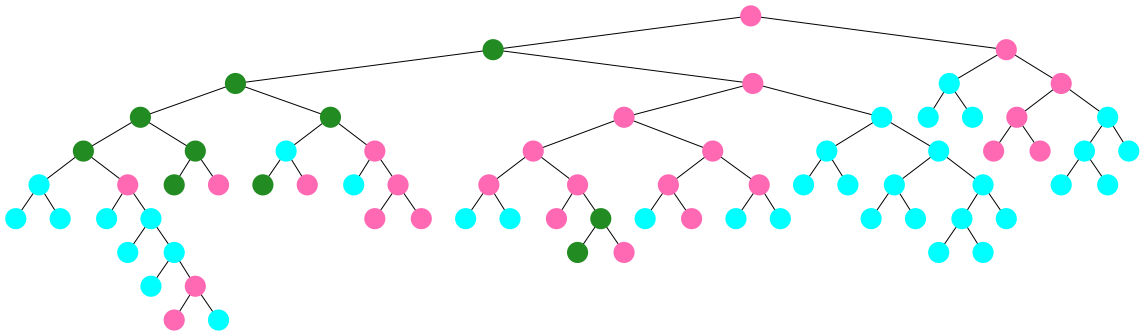

Figure 4: Parallel ABACUS, Hybrid Strategy (Tolerance=2%), 74 nodes.

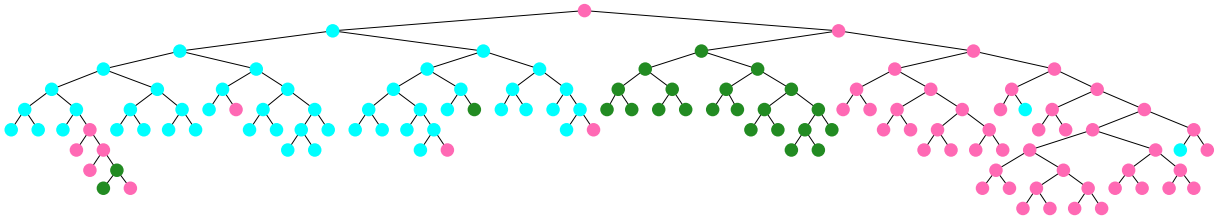

Figure 5: Parallel ABACUS, Local Best First Strategy, 125 nodes.

References

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

[2] 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/distribution.html.

[3] M. Böhm. Parallel ABACUS – Implementation. Technical Report 99.359, Institut für Informatik, Universität zu Köln, 1999.

[4] M. Böhm. Parallel ABACUS – Software Distribution, 1999. http://www.informatik.uni-koeln.de/ls_juenger/projects/abacus/parallel.html.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable Object–Oriented Software*. Addison Wesley Longman, 1995.

[6] S. Leipert. The Tree Interface – Version 1.0 user manual. Technical Report 96.242, Institut für Informatik, Universität zu Köln, 1996. http://www.informatik.uni-koeln.de/ls_juenger/projects/vbctool.html.

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

[8] D. C. Schmidt. Active Object, An object behavioral Pattern for Concurrent Programming. In Greg Lavender, editor, *Pattern Languages of Programming Design 2*. Addison Wesley, MIT Press, 1996.

[9] S. Thienel. *ABACUS — A Branch–And–CUt System*. PhD thesis, Universität zu Köln, 1995. http://www.informatik.uni-koeln.de/ls_juenger/publications/thienel/diss.html.

[10] S. Thienel. A simple TSP–solver: An ABACUS tutorial. Technical Report 96.245, Institut für Informatik, Universität zu Köln, 1996.