A Work on the Boundary Integral Solver for the Parallel Particle Mesh Library

&

An Implementation of the CHARMS Framework

Diploma thesis at ICoS¹

Ingo Oppermann
ingo@student.ethz.ch
Rechnergestützte Wissenschaften

July 30, 2004

Advisers: Ivo F. Sbalzarini
Dr. Jens H. Walther
Michael Bergdorf
Prof. Petros Koumoutsakos

¹Institute of Computational Science, Department of Computer Science, Swiss Federal Institute of Technology (ETH), Zürich, http://www.icos.ethz.ch/
Contents

1 Introduction .................................................. 1

2 PPM ......................................................... 2
   2.1 Ring topology ........................................ 2
      2.1.1 Invoking the ring topology .................... 3
      2.1.2 Using the ring topology ....................... 3
      2.1.3 Test case ...................................... 3
      2.1.4 Routines ...................................... 5
   2.2 Connections ........................................ 8
      2.2.1 Using connections .............................. 9
      2.2.2 Test case .................................... 11
      2.2.3 Routines .................................... 12
   2.3 Boundary integrals .................................. 15

3 CHARMS .................................................. 19
   3.1 Using CHARMS .................................... 19
      3.1.1 Geometry data ............................... 20
      3.1.2 Boundary conditions ......................... 21
      3.1.3 Refinement levels ............................ 22
      3.1.4 Calculation .................................. 22
      3.1.5 Basis (un)refinement ....................... 25
   3.2 Applications ...................................... 25
      3.2.1 1D Poisson equation ......................... 25
      3.2.2 2D Poisson equation ......................... 28
      3.2.3 2D Heat equation ............................ 29
      3.2.4 Heat equation on a sphere .................. 30

4 Conclusions ........................................... 35

5 Outlook ................................................. 35

6 References ............................................ 36

A CHARMS API Reference .................................. 37
   A.1 Initializing ....................................... 37
   A.2 Destroying ....................................... 37
   A.3 Adding nodes .................................... 37
   A.4 Adding elements .................................. 37
   A.5 Adding boundary conditions .................... 38
   A.6 Building the refinement levels ................. 39
   A.7 Building the integration elements ............. 39
   A.8 Refining a basis .................................. 39
   A.9 Unrefining a basis ................................ 40
   A.10 Setting the weights .............................. 40
   A.11 Getting the solution at a node ............... 40
   A.12 Setting the weights and getting the solution . 41
   A.13 Verbose output .................................. 41
   A.14 Evaluating basis functions .................... 41
B CHARMS Implementation details 42
B.1 Compiling 42
B.2 Naming conventions 42
B.3 Data structures 42
1 Introduction

The first project of this thesis is to implement capabilities to the Parallel Particle Mesh Library (PPM) in order to have the tools for solving boundary integrals. On the way to that there is the ring topology in order to have the ability of doing all-with-all particle interactions or just sending arbitrary data from one processor to another. The next step are the connections. These enables the user to define relationships between particles. Such a connection can contain an arbitrary number of particles that can be seen as molecules or elements of a boundary depending on the user’s intent.

The second project is an implementation of the "Conforming, Hierarchical, Adaptive Refinement Methods" (CHARMS) framework. This a method for finite element solvers to do adaptive refinements not on the elements, but on the basis functions and leads to nested hierarchical sets of meshes and bases. Then a basis function can be expressed as a linear combination of basis functions on the next finer refinement level that support the same element. This avoids complicated strategies for keeping a refined geometry consistent and is much simpler to implement.
2 PPM

In this section I describe my contribution to the Parallel Particle Mesh Library (PPM). These are the ring topology, the connection handling and some helper routines for calculating boundary integrals.

2.1 Ring topology

The ring topology provides the capability to the PPM to do computations where every particle has to interact with every other particle in the system. This means, since every particle has to interact with every other particle, a decomposition of the computational domain based on the particle positions and a cut-off radius is not relevant anymore. With this in mind every available processor gets the same given domain and the particles are mapped to the processors with no respect to their position in the computational space. Each processor gets as many particles as it can handle according to the performance (i.e. if all processors perform equally the particles will be distributed evenly among them).

The notion “ring topology” has its origin in the numbering of processors in the current hardware setup. With this numbering one can define a neighborhood of a processor with rank \( k \) where the left neighbor is the one with rank \( k - 1 \) and the right neighbor is the one with rank \( k + 1 \). The ranks of the neighbors has to be taken modulo the total number of processors in the setup in order to get the ring. This way if every processor sends some data to its left neighbor and receives some data from its right neighbor for \( n \) times (where \( n \) is the number of processors) every processor will finally get back its own data.

In order to perform a calculation on the ring, a copy of the local data has to be send around the ring such that the local particles “sees” every copy of the other processors. For the non-symmetric case the copy particles are send around the whole ring. To speed this up, the interactions can be calculated symmetrically (actio-reactio) where the copy is only send around the half way and then send back where it originated from. Special care must be taken if the total number of processors is even because the two opposite processors will both calculate the same interactions which is once too much. To get around that, both processors compute only one half of the interactions each. For more details on that please refer to the implementation details (section 2.1.4).

Being able to send some data around the ring is not only useful for interaction calculations but also to distribute undirected information among the processors. Undirected means, if the sender or the receiver of this information is unknown. For example if every processor needs particles that are not locally available and has particles that may be needed by other processors but does not know who has or needs them, a copy of the local particles can be send around the ring and each processor can check the just arrived copy from the neighbor for its needed particles to pick out. After one round every processor has its particles to calculate with. If a result of an interaction with a particle of an other processor has to be send back one has to keep track of where the copy particle came from. Would it be that this information is not stored the distribution problem can also be solved if every processor broadcasts its information or calls the MPI_AllToAll(….) routine, but this may be a too big impact on the network, especially if this has to be done every time step of a simulation.
2.1.1 Invoking the ring topology

Because any kind of decomposition of the domain, respectively of the particles, is a valid ring topology, there exists only one such topology and therefore only one unique ID for this topology is necessary. The ring topology has always the special ID 0 (zero).

First, one has to create the ring topology with a call to `ppm_maketopo(...)`. The parameter that defines the decomposition must be `ppm_param_decomp_null`, the so-called null decomposition. This takes care that every processor gets the same user-defined computational domain (every processor will have one subdomain that is equal to the domain) and the topology ID will be set to 0, if not already. The assignment parameter does not have any effect. Any user-defined domain decomposition will be ignored.

For the mapping of the particles to the ring topology one has to call the `ppm_map_part(...)` routine with topology ID equals to 0 and the mapping type `ppm_param_map_global`. This will call the `ppm_map_part_ring(...)` routine that finds out where to send to or receive from particles and fills the internal data structures accordingly. For further data to send along with the particles one has to use the push and pop functionality of this routine.

2.1.2 Using the ring topology

The calculation of the interactions on the ring is demonstrated in the two provided templates `ppm_template_ring_interaction.f` and `ppm_template_comp_ring_pp.f` that can be found in the src/ directory of the library. The former contains the code for sending the copies of the local data around and calls to the latter template that actually do the computations. These templates are intended to be edited by the user to fit his/her special needs. Inside them there are a lot of comments which explain what is going on and hints where to modify or to add some code in order to customize these templates.

The main aspect of the ring topology is to send data around. The routine `ppm_map_part_ring_shift(...)` is designed for that. It takes the target and the source rank (e.g. left and right neighbor) and the data to be send as parameter. Because the target and source processor have to be given explicitly, this routine cannot be called through `ppm_map_part(...)` but it is an interface to the push-send-pop functionality of the `ppm_map_part(...)` routine. It sets up the send and receive lists and buffers such that all the provided data will be send to the processor with the target rank and received from the processor with the source rank. The ring shift is actually not limited to the ring topology. It can also be used to send data around in other topologies without modifying them. Since the ring shift is independent of the current topology the parameter for the topology ID in subsequent calls to `ppm_map_part(...)` can always be set to -1 or any other negative value.

2.1.3 Test case

In order to test the ring topology I wrote a Lennard-Jones molecular dynamics application based on the existing one that uses cell lists to calculate the interactions and modified it such that it uses the ring topology instead. The particle positions are read in from a file such that the processor with rank 0 gets so much
Figure 1: Oscillation of total energy over 3 simulated seconds with different time step sizes. Calculated on one processor using symmetry. Left: step size $\Delta t = 0.01s$. Right: step size $\Delta t = 0.001s$

particles that each other processor gets only one particle. Then the ring topology gets created and the particles are mapped to it. After that the particles are distributed evenly among the processors. Inside the time step loop there is no new mapping performed since there is no need to move particles from one processor to another in the ring topology. The integration of the equations of motion is done using the velocity Verlet algorithm [1]. The used potential is

$$V_{LJ}(r_{ij}) = -4\epsilon \left( \frac{\sigma}{r_{ij}} \right)^6 - \left( \frac{\sigma}{r_{ij}} \right)^{12}$$

with $\epsilon = 1$ and $\sigma = 0.5$.

The simulation environment are periodic boundary conditions. This kind of simulation is useful to test if the total energy in the system stays constant over time. The computational domain has an extent of 100x100x100 units with one thousand randomly placed particles with no initial velocity in it. The simulation time is 3 seconds with step sizes of $\Delta t = 0.01s$ (300 time steps) and $\Delta t = 0.001s$ (3000 time steps). The number of processors vary from 1 to 4 and for each time step size the system is calculated with and without the use of symmetry.

The initial total energy is for all configurations with $-0.5905970281$ the same. There are fluctuation in this value for the different configurations, but they take place at an order of $10^{-15}$ and are due to truncation errors by the floating point arithmetic. During the evolution of the system the total energy oscillates around that value. The amplitude of the oscillation depends on the time step size where the oscillation gets smaller with smaller time step sizes. In the table below the minimal and the maximal total energies are listed.

<table>
<thead>
<tr>
<th>$\Delta t$</th>
<th>min. energy</th>
<th>max. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01s</td>
<td>-0.6218607</td>
<td>-0.5548341</td>
</tr>
<tr>
<td>0.001s</td>
<td>-0.5909240</td>
<td>-0.5902516</td>
</tr>
</tbody>
</table>

Table 1: Oscillation of total energy with different time step sizes.

The changes in the total energy are of an order of $10^{-1}$ for a step size of 0.01s.
in contrast to $10^{-4}$ for a step size of 0.001 s. Figure 1 shows the fluctuation of the total energy during the whole simulation with both time step sizes calculated on one processor using symmetry.

### 2.1.4 Routines

In this section I describe the routines that I wrote in order to use the ring topology. First I start with the two mentioned templates and the `ppm_map_part_ring_shift(...)` routine. Then I explain what the internal (i.e., not directly accessible by the user) routines `ppm_decomp_null(...)` and `ppm_map_part_ring (...)` are doing.

**SUBROUTINE ppm_template_ring_interaction(xp, vp, Npart, &
lsym, fp, params, &
info)**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp(:, :)</td>
<td>F</td>
<td>The particle coordinates</td>
</tr>
<tr>
<td>vp(:, :)</td>
<td>F</td>
<td>Additional particle data used for interactions</td>
</tr>
<tr>
<td>Npart</td>
<td>I</td>
<td>The number of particles</td>
</tr>
<tr>
<td>lsym</td>
<td>L</td>
<td>Calculate with or without symmetry</td>
</tr>
<tr>
<td>fp(:, :)</td>
<td>F</td>
<td>Change of particle data due to interaction</td>
</tr>
<tr>
<td>params(;)</td>
<td>F</td>
<td>User defined parameter and/or output of interaction (e.g., potential energy)</td>
</tr>
<tr>
<td>info</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This is the main routine for calculating the particle-particle interactions on the ring. This one gets called in the user’s program. The parameter list is not fixed and should be modified by the user according to what has to be calculated. The neighborhood of the processors can be easily computed. Let $n$ be the number of processors in the setup and $k$ will denote the rank of a single processor and can take values from 0 to $n-1$. Then the left (target) neighbor is $k-1 \bmod n$ and the right (source) neighbor is $k+1 \bmod n$. How often the data has to be shifted around depends on the use of symmetry. For the non-symmetric case the results of the interaction (e.g., forces) are accumulated to the local data or to the current copy. For that the copies have to be send around the whole ring in order to get all interactions. Thus the number of shifts (hops) is $h = n-1$. After this number of hops every particle has interacted with each other. Then only a last hop is needed such that the data is there where it originated from. To speed this up, the interactions can be calculated symmetrically (actio-reactio) where the result is accumulated to the local ones and to the current copy with opposite sign. This requires the copy to only send around the half way and then to send it back where it originated from. But this requires to distinguish between the two cases if $n$ is odd or even. In the simpler case where $n$ is odd the number of hops is $h = ((n+1)/2)-1$ and then the data can be send back. In the even case it is more tricky. The number of hops is $h = (n/2)-1$. If the data will then be send back, not all interactions would have been considered. One hop more is done and one processor calculates one half, i.e., all its local particles interact with the lower half of the copy particles (which are the lower half of the local particles of the opposite processor), and the other one calculates the other half, i.e., the upper half of its local particles interact
with all copy particles (which are all local particles of the opposite processor). After that every processor gets its copy back and add up these result to the local ones. For sending back the results the target is \( k + (n/2) \mod n \) and the source is \( k - (n/2) \mod n \).

**SUBROUTINE ppm_template_comp_ring_pp**(xp, vp, fp, Npart, &xp2, vp2, fp2, Lpart, &lsymm, params, mode, info)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp(i,:),</td>
<td>F</td>
<td>The particle coordinates (first group)</td>
</tr>
<tr>
<td>vp(i,:),</td>
<td>F</td>
<td>Additional particle data used for interactions (first group)</td>
</tr>
<tr>
<td>fp(i,:),</td>
<td>F</td>
<td>Change of particle data due to interaction (first group)</td>
</tr>
<tr>
<td>Npart,</td>
<td>I</td>
<td>The number of particles (first group)</td>
</tr>
<tr>
<td>xp2(i,:),</td>
<td>F</td>
<td>The particle coordinates (second group)</td>
</tr>
<tr>
<td>vp2(i,:),</td>
<td>F</td>
<td>Additional particle data used for interactions (second group)</td>
</tr>
<tr>
<td>fp2(i,:),</td>
<td>F</td>
<td>Change of particle data due to interaction (second group)</td>
</tr>
<tr>
<td>Lpart,</td>
<td>I</td>
<td>The number of particles (second group)</td>
</tr>
<tr>
<td>lsymm,</td>
<td>L</td>
<td>Calculate with or without symmetry</td>
</tr>
<tr>
<td>params(:),</td>
<td>F</td>
<td>User defined parameter and/or output of interaction (e.g. potential energy)</td>
</tr>
<tr>
<td>mode,</td>
<td>I</td>
<td>Interaction mode (first group is (not) the same as the second group)</td>
</tr>
<tr>
<td>info,</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This is the routine where the actual interaction between two groups of particles (i.e. the local particles and the copy particles) takes place. This template gets called by the `ppm_template_ring_interaction.f` template. The parameter list is not fixed and should be modified by the user according to what has to be calculated. Here two modes have to be distinguished. If both groups contain the same data (i.e. \( xp2 \) is the copy of \( xp \) and so on) then for the symmetric case only the interaction between unique particle pairs has to be considered. For the non-symmetric case only the interaction of a particle with itself is skipped. If the two groups have different data (i.e. \( xp \) is the local data and \( xp2 \) is the copy data from another processor) then in both cases, the symmetric and non-symmetric, it is iterated over all particle pairs.

**SUBROUTINE ppm_map_part_ring_shift**(xp, lda, Npart, &itarget, isource, &info)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp(i,:),</td>
<td>F</td>
<td>The particle coordinates</td>
</tr>
<tr>
<td>lda,</td>
<td>I</td>
<td>The leading dimension of xp</td>
</tr>
<tr>
<td>Npart,</td>
<td>I</td>
<td>The number of particles</td>
</tr>
<tr>
<td>itarget,</td>
<td>I</td>
<td>The ID of the processor to send to</td>
</tr>
<tr>
<td>isource,</td>
<td>I</td>
<td>The ID of the processor to receive from</td>
</tr>
<tr>
<td>info,</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>
This routine prepares the send and receive lists and the buffers for sending the particles to the processor with rank itarget and for receiving the particles from the processor with rank isource. After the call of this routine one can push additional data to the buffers with calls to ppm_map_part(…, ppm_param_map_map_push,…). Sending and retrieving the new data is done with the same routine but with the according parameter ppm_param_map_send and ppm_param_map_map_pop. This routine is independent of the current topology ID and will not change it in any way.

SUBROUTINE ppm_decomp_null(min_phys,max_phys, &
minimum, maximum sub-domains, &
subs, info)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>min_phys(:)</td>
<td>F</td>
<td>The minimum coordinates of the physical / computational domain</td>
</tr>
<tr>
<td>max_phys(:)</td>
<td>F</td>
<td>The maximum coordinates of the physical / computational domain</td>
</tr>
<tr>
<td>min_sub(:,;:)</td>
<td>F</td>
<td>The minimum extent of the sub domain</td>
</tr>
<tr>
<td>max_sub(:,;:)</td>
<td>F</td>
<td>The maximum extent of the sub domain</td>
</tr>
<tr>
<td>subs</td>
<td>I</td>
<td>The total number of sub domains</td>
</tr>
<tr>
<td>info</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This routine performs a trivial decomposition of the computational domain into one sub domain. This one sub domain has the same size as the computational domain. subs is set to 1 and the contents of min_phys and max_phys gets copied into the newly allocated arrays min_sub(1,:) and max_sub(1,:).

SUBROUTINE ppm_map_part_ring(xp,Npart,info)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>xp(:,;:)</td>
<td>F</td>
<td>The particle data</td>
</tr>
<tr>
<td>Npart</td>
<td>I</td>
<td>The number of particles</td>
</tr>
<tr>
<td>info</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This routine maps the particles onto the given processors without respect to the topology, i.e. every processor gets as many particles as it can handle according to the processor speed. The processor speed is stored in a global variable called ppm_proc_speed and contains the relative speeds that adds up to 1. To calculate the final number of particle per processor the total number of particles in the system and how many particles are on each other processor is determined with a call to MPI_AllReduce(…). The particles per processor are stored in the array plist_excl. The array plist_dex contains the desired number of particles per processor. This is computed out of the total number of particles and the information in ppm_proc_speed. Unassigned particles (due to truncation errors) are evenly added to the desired list. Now the excess, the difference between the actual and the desired number of particles per processor, is stored in the array plist_exc. The index of these arrays corresponds to the ranks of the processors. A positive excess means that this processor has too much, and a negative excess means that this processor has too few particles. Based on this information, the excess gets reduced to 0 on all processors. First
the two processors with the minimal and the maximal excess has to be found. If two or more processors have the same minimal or maximal excess then the ones with the lowest rank are taken. Then the processor with maximum excess has to give as many particles as it can to the other processor if the absolute value of the minimal excess is bigger or equal to the maximal excess. Otherwise only as much particles as needed has to be given to the other processor. Every processors does this calculation but only keeps track of how many particle it has to send to whom or to receive by whom if it is one of the two processors with the minimal or maximal excess. This information is stored in the two lists slist1 and slist2. The former list stores the rank of the receiver or of the processor to receive from. The latter list contains the number or particles to send or receive. A processor has only either to send or receive particles or to do nothing if the excess of this one already zero. The reduction of the excess has an end as soon as the ranks of the processors with minimal and maximal excess are equal. With this information at hand the send and receive lists and the buffer can be set up in order to use the ppm_map_part(...) interface to transfer the data.

2.2 Connections

Connections define a relationship between two or more particles. As the name suggests all particles in a connection have (at least) one thing in common. This may be that they are in the same molecule or they define the elements of a boundary and so on.

Each connection consists of a list of the IDs of the particles that are in it. Such a list can be of any length. Since a connection does not carry any geometric information, i.e. a connection with three particles can represent a triangle where every particle is connected with every other or a molecule with three atoms where only the first and the second, the second and the third particle are connected. This interpretation depends on the user’s intent.

The first step after reading in the connection data, is to distribute them to the processors where the particles of the connection reside. After the distribution, every processor that has at least one particle in a connection has the whole connection. In this situation if a connection consists of three particles and these particles are on three different processors then the connection information is on every of these three processors available. Then all three processors would compute the connection specific interaction. To circumvent this a decision has to be made what processor has to calculate the connection specific interaction. The implemented criterion is that only the processor which has the particle with the lowest global ID in the connection keeps the connection. All other processors discard this connection. If symmetry is used for calculation then only the processor who has all the particles of a connection keeps the connection. This presupposes that the length of the connection is smaller than the size of the ghost layer around the sub domain because some particles in the connection may be particles of an other sub domain/processor. If the ring topology is used, the distinction between symmetry or not is not necessary. There the calculation of the connection interaction does not depend on it.

Since a processor only has a connection where it has the particle with the lowest ID of (not using symmetry or using the ring topology), it may occur that not all particles are locally available. For that, the missing data (particle
positions, velocities, etc.) that is needed for calculating the interactions in the connection has to be collected. Then an iteration through all connections can be performed in order to get the interactions.

### 2.2.1 Using connections

The interface to the connections handling is given by the `ppm_map_connect(...)` routine. It takes the connection data that is stored in a 2 dimensional array where the leading dimension correspond to the length of the connection. This leads to the condition that only connections which have the same length can be passed to the routine per call. Connections that have a different length must be stored in a different array. Every connection consists of a list of global IDs of the particles that represent the connection. This is why the mapping information from local particle IDs to global particle IDs has always to be passed along with the connection data. Then three types of actions define what to do with the provided connections. `ppm_param_connect_distribute` indicates that the given connections should be distributed to the processors that have particles of them and this should be the first thing to be done after the particles have been mapped to the processors. The contents of the connection data array gets replaced by the remaining and new connections. Depending on how the connections have been read in it may occur that some connections are stored locally more than once or there are no particles to a connection (orphan connection). These connections are removed to retain consistency. `ppm_param_connect_prune` indicates that the given connections should be checked for unneeded ones based on the used topology and the use of symmetry. `ppm_param_connect_send` is for sending the connections according to the current mapping introduced with a call to `ppm_map_part(...)` with the parameters `ppm_param_map_global` or `ppm_param_map_partial`. Will say, if a particle will be moved to another processor the connection containing this particle will also be copied to that processor. Here, again it may occur that some connections are stored locally more than once or there are orphan connections. These connections are removed to retain consistency.

The use of the connection handling interface can be explained most simply on the basis of an example. Consider three processors with 5 particles distributed among them and three connections of length 3. The connections are all read in by the first processor. The situation is as follows:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Particles</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2</td>
<td>1 - 2 - 3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2 - 4 - 5</td>
</tr>
<tr>
<td>3</td>
<td>4, 5</td>
<td></td>
</tr>
</tbody>
</table>

First a call to `ppm_map_connect(...)` with the parameter `ppm_param_connect_distribute` as the action to be performed distributes the connections to the other processors:
<table>
<thead>
<tr>
<th>Processor</th>
<th>Particles</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2</td>
<td>1 – 2 – 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – 4 – 3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1 – 2 – 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – 4 – 3</td>
</tr>
<tr>
<td>3</td>
<td>4, 5</td>
<td>1 – 4 – 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 – 4 – 5</td>
</tr>
</tbody>
</table>

Now the pruning (ppm_param_connect_prune) of the unneeded connections must be done. In this example only the connections will be kept where the particle with lowest ID is locally available (as for non-symmetry or the ring topology):

<table>
<thead>
<tr>
<th>Processor</th>
<th>Particles</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, 2</td>
<td>1 – 2 – 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – 4 – 3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3 – 4 – 5</td>
</tr>
</tbody>
</table>

To demonstrate the effect of the ppm_param_connect_send action type consider that a new mapping leads to a new particle distribution shown in the following table. Since this new mapping has been introduced by a call to ppm_map_part(...) the send list contains the information about where what particles are going to be send to whom. If two particles of a connection will be send to another processor the connection will only be send once. If each of the two particles will be send to different processors the connection will be send to both processors.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Particles</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1 – 2 – 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – 4 – 3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3 – 4 – 5</td>
</tr>
</tbody>
</table>

A call of ppm_map_connect(...) with the send action type right after the mapping but before actually sending the particles will pick out these informations and send the connection accordingly and will look as follows (orphan connections are already removed):

<table>
<thead>
<tr>
<th>Processor</th>
<th>Particles</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1 – 2 – 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – 4 – 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 – 4 – 5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1 – 2 – 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 – 4 – 3</td>
</tr>
<tr>
<td>3</td>
<td>2, 4, 5</td>
<td>1 – 2 – 3</td>
</tr>
</tbody>
</table>

After the sending of the particles is performed and every information has been popped out of the receive buffer (especially the new local to global ID mapping) the unneeded connections have to be pruned again:
<table>
<thead>
<tr>
<th>Processor</th>
<th>Particles</th>
<th>Connections</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3 - 4 - 5</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1 - 2 - 3</td>
</tr>
<tr>
<td>3</td>
<td>2, 4, 5</td>
<td>1 - 4 - 3</td>
</tr>
</tbody>
</table>

For the connection handling there is still a lot within the responsibility of the user. He/she has to keep track of the mapping of the particle IDs or the sending and pruning of the connections has to be done at the right time at the right place.

For calculating the interactions within the local connections a template called `ppm_template_connect_interaction.f` can be found in the `src/` directory of the PPM library. It cares about collecting the missing data of the connections (if needed) and about sending back the results where they belong to. In between the actual computation of the interactions takes place. This template is intended to be edited by the user to fit his/her special needs. Inside it there are a lot of comments which explain what is going on and hints where to modify or to add some code in order to customize this template. Please refer to the implementation details for more detailed description of what operations are performed (section 2.3.3).

### 2.2.2 Test case

This test case is similar to the test case for the ring topology. Additional to the Lennard-Jones potential, pairs of particles are influenced by a spring force (Hooke’s law). The particle pairs are defined by connections of length 2. The extent of the domain is the same, but only 500 particles are placed randomly in it. The positions of the particles are generated such that every particle has a partner that it 5 units away. These 5 units are the reference length for the spring force. The integration scheme, the parameter for the Lennard-Jones potential and the boundary conditions are the same as in the test case for the ring topology. Again the total energy of the system should stay constant over time. The used potential for the spring force is

\[ V_S(r_{ij}) = \frac{k}{2} (r_0 - r_{ij})^2 \]

with \( k = 0.01 \) as the spring constant and \( r_0 = 5 \). The simulation time is 10 seconds with time step sizes of \( \Delta t = 0.01s \) and \( \Delta t = 0.001s \). This is calculated on 1 to 4 processors with symmetry. For calculation the connection interaction on the ring topology it does not make a difference it symmetry is used or not.

The initial total energy for all configurations is \(-0.3450196404\) and the difference between the use of one and more processors is of the order of \(10^{-15}\). The minimal and maximal total energy at the two time step sizes are shown in the table below.

<table>
<thead>
<tr>
<th>( \Delta t )</th>
<th>min. energy</th>
<th>max. energy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01s</td>
<td>-0.3761218</td>
<td>-0.3151287</td>
</tr>
<tr>
<td>0.001s</td>
<td>-0.3453385</td>
<td>-0.3447479</td>
</tr>
</tbody>
</table>

Table 2: Oscillation of total energy with different time step sizes.
The changes in the total energy are of an order of $10^{-2}$ for a step size of 0.01s in contrast to $10^{-3}$ for a step size of 0.001s. Figure 2 shows the fluctuation of the total energy during the whole simulation with both time step sizes calculated on one processor.

To see if the interactions of the connection have a certain influence on the system, figure 3 shows the difference in the evolution of the total energy of the system with and without connections.

![Figure 2: Oscillation of total energy over 10 simulated seconds with different time step sizes. Calculated on one processor. Left: step size $\Delta t = 0.01s$. Right: step size $\Delta t = 0.001s$.](image)

![Figure 3: Difference in total energy of the system with and without connections calculated on one processor. 10 simulated seconds with a time step size $\Delta t = 0.01s$.](image)

### 2.2.3 Routines

In this section I describe the routines that I wrote in order to use the connection mapping. First I start with the mentioned template and the `ppm_map_connect(…)` routine. Then I explain what the internal (i.e. not directly acces-
sible by the user) routines \texttt{ppm\_map\_connect\_distrib(...), ppm\_map\_connect\_send(...)} and \texttt{ppm\_map\_connect\_prune(...)} are doing.

\textbf{SUBROUTINE} ppm\_template\_connect\_interaction(id,xp,vp,Npart & cd,ldc,Ncon & 1loc,fp,params, & info)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id(:)</td>
<td>I</td>
<td>Local to global particle number mapping</td>
</tr>
<tr>
<td>xp(:, :)</td>
<td>F</td>
<td>The particle coordinates</td>
</tr>
<tr>
<td>vp(:, :)</td>
<td>F</td>
<td>Additional particle data used for interactions</td>
</tr>
<tr>
<td>Npart</td>
<td>I</td>
<td>The number of particles</td>
</tr>
<tr>
<td>cd(:, :)</td>
<td>I</td>
<td>The connections</td>
</tr>
<tr>
<td>ldc</td>
<td>I</td>
<td>The length of the connections</td>
</tr>
<tr>
<td>Ncon</td>
<td>I</td>
<td>The number of connections</td>
</tr>
<tr>
<td>1loc</td>
<td>L</td>
<td>All informations locally available or not</td>
</tr>
<tr>
<td>fp(:, :)</td>
<td>F</td>
<td>Change of particle data due to interaction</td>
</tr>
<tr>
<td>params(;)</td>
<td>F</td>
<td>User defined parameter and/or output of interaction (e.g. potential energy)</td>
</tr>
<tr>
<td>info</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This is the routine for calculating the interactions in a connection. This one gets called in the user's program. The parameter list is not fixed and should be modified by the user according to what has to be calculated or what is needed for the calculation. First all particles that are in the local connections have to be gathered. If not all informations are locally available they have to be collected from the other processors. Since it is not known how the particles are distributed on the other processors, each processor sends a copy of its particles (and further information if needed) along the whole ring such that each processor can pick out the particles it needs for its connections. In order to be able to send any results back that affect particles that are from another processor it has to be stored where it belongs to. The container for this is the \texttt{pmap} array which has three entries for each particle that is involved in a connection. The first entry is the rank of the processor it belongs to. The second one is the local ID of the particle on that processor and the last one holds the global ID of the particle. The particles that are going to be used for the interaction calculation are stored in an extra array called \texttt{xp\_con}, \texttt{vp\_con} and so on. If a particle is in more than one connection it is stored only once and every particle gets a new local ID. With the information stored in \texttt{pmap} the global particle IDs in the connection array \texttt{cd} gets translated to the new local IDs and are stored in \texttt{cd\_local}. Then the access to the particle data of the connections is quite simple. For example the IDs of the particles of the second connection can be accessed by \texttt{p(:,)=cd\_local(:,2)}. Then the coordinates of the particles are \texttt{xp\_con(:,p(1)), xp\_con(:,p(2))} and \texttt{xp\_con(:,p(3))} for a connection of length 3. Then the results can be stored in the \texttt{fp\_con} array. The last step is to add the results to the existing ones. The results for the other processors are send around the ring again. The \texttt{pmap} array has to be send along too such that the other processors know where the results belong to.

\textbf{SUBROUTINE} ppm\_map\_connect(cd,lda,Ncon,ld,aspect, &
This routine is the interface to the different connection handling routines. It does nothing more than calling the appropriate routine according to the connection handling type. The known types are `ppm_param_connect_distribute` for distributing the connections, `ppm_param_connect_prune` for pruning the connections and `ppm_param_connect_send` for sending the connections based on the current particle mapping.

**SUBROUTINE ppm_map_connect_distrib(cd,lda,Ncon,id, & Npart, info)**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cd(:, :)</td>
<td>I</td>
<td>The connections</td>
</tr>
<tr>
<td>lda</td>
<td>I</td>
<td>The length of the connections</td>
</tr>
<tr>
<td>Ncon</td>
<td>I</td>
<td>The number of connections</td>
</tr>
<tr>
<td>id(:, :</td>
<td>I</td>
<td>Local to global particle number mapping</td>
</tr>
<tr>
<td>Npart</td>
<td>I</td>
<td>The number of particles</td>
</tr>
<tr>
<td>contype</td>
<td>I</td>
<td>Connection handling type</td>
</tr>
<tr>
<td>lsymm</td>
<td>L</td>
<td>Using symmetry or not</td>
</tr>
<tr>
<td>info</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This routine distributes the given connections to the processors such that every processor has the connection where it has at least one particle for. The particle IDs in the connections get compared against the IDs of the given local available particles. Only these connections are kept where at least on particle is locally present. This also automatically removes orphan connections. Then the connection data is send around the ring such that every processor can pick out the connections it has particles for. After one round over processor has all needed connections. Since the connection data consists only of integers they are send as such by directly calling the `MPI_SendRecv(...)` routine and not with the `ppm_map_part(...)` interface to avoid type conversions to REAL and back to INTEGER. The connections that are stored more than once are filtered out by comparing each connection with each other with respect to the order the IDs are listed in a connection (e.g. the connection $1 - 2 - 3$ is not the same as $1 - 3 - 2$ or any other permutation). Redundant connections get marked as such and will be skipped when arranging the final set of connections.

**SUBROUTINE ppm_map_connect_prune(cd,lda,Ncon,id,Npart, & lsymm, info)**
This routine removes the connections that are not needed on this processor. Two modes of pruning can be performed. The first mode keeps only those connections where the particle with lowest ID is locally available. The minimum ID of the connections is taken and looked up in the id array. If it can be found this connection stays. This mode is applied if the system is not symmetric or if the current topology is the ring topology. The second mode keeps only those connection where all particles in a connection are locally available. All IDs in a connection are looked up the in id array. If the number of found IDs equals the length of the connection then this connection stays. This is applied if the system is symmetric and the current topology is not the ring topology.

SUBROUTINE ppm_map_connect_send(cd, lda, Ncon, id, Npart, info)

This routine sends the connections based on the current mapping. It reads out the informations stored in the ppm_sendbuffer and ppm_buffer2part arrays in order to find out what particles will be send away. The affected connections are marked in a destination-connection map called psend. The destination and the source of the connections are read out of the ppm_isendlist and ppm_irecvlist arrays. Before actually sending the connection data, the number of connections to send and receive will be exchanged. Since the connection data consists only of integers they are send as such by directly calling the MPI_SendRecv(...) routine and not with the ppm_map_part(...) interface to avoid type conversions to REAL and back to INTEGER. Therefore a call of this routine does not interfere with ppm_map_part(...). After sending and receiving the connections, multiple and orphan connections are removed.

2.3 Boundary integrals

The functions in the BEM module ppm_module_bem provides some tools to carry out numerical integrations over a triangle. These tools consist of a set of quadrature rules that can be easily extended and some interpolation schemes (constant, linear and quadratic). This module is used in the template ppm_template_comp_boundary.f that calculates the interaction of the particles with the boundary elements and can be found in the src/ directory. This template make use of
the connections since the boundary elements are assumed as triangles defined as
connected particles. The interaction of the particles with the boundary elements
is done such that every local connection first interact with every local particle
and the particles are send around the ring and interact with the connections of
every other processor. With that, every particle finally interacted with every
connection or boundary element. Unfortunately the module and the template
are not tested due to a lack of time.

In the following are the descriptions of the routines of the module and the
template.

SUBROUTINE ppm_bem_quadrule_npoints(rule,nqp,info)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rule</td>
<td>I</td>
<td>The quadrature rule</td>
</tr>
<tr>
<td>nqp</td>
<td>I</td>
<td>The number of quadrature points</td>
</tr>
<tr>
<td>info</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This routine returns the number of quadrature points for a given quadrature
rule. The available rules are

<table>
<thead>
<tr>
<th>Rule</th>
<th>Points</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppm_param_bem_quadrule_center</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ppm_param_bem_quadrule_nodes</td>
<td>3</td>
<td>?</td>
</tr>
<tr>
<td>ppm_param_bem_quadrule_edges</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>ppm_param_bem_quadrule_cne</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>ppm_param_bem_quadrule_stroud</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

These rules are described in the description of the routine ppm_bem_quadrule
(...) that actually returns the quadrature points and its weights.

SUBROUTINE ppm_bem_quadrule(qp,qw,rule,info)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qp(:, :)</td>
<td>F</td>
<td>The quadrature points</td>
</tr>
<tr>
<td>qw(:, )</td>
<td>F</td>
<td>The weights</td>
</tr>
<tr>
<td>rule</td>
<td>I</td>
<td>The quadrature rule</td>
</tr>
<tr>
<td>info</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This routine returns the quadrature points and their weights according to the
given rule. The arrays for the points and weights have to be big enough to hold
these. The leading dimension of qp is always 2. The quadrature points lie all in
the unit triangle and the weights sum up to 1. ppm_param_bem_quadrule_center
denotes the center point of the triangle as in [2]. ppm_param_bem_quadrule_ nodes
are the corners of the triangle, ppm_param_bem_quadrule_edges are the
midpoints of the edges (found in [3]) and ppm_param_bem_quadrule_cne is the
combination of the former three. The last rule is if the book by A. Stroud (see
[4], T25-1).

SUBROUTINE ppm_bem_basis(qp,lb,basis,info)
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>qp(:, :)</td>
<td>F</td>
<td>The point to be interpolated</td>
</tr>
<tr>
<td>lb( :)</td>
<td>F</td>
<td>The basis coefficients</td>
</tr>
<tr>
<td>basis</td>
<td>I</td>
<td>The interpolation scheme</td>
</tr>
<tr>
<td>info</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This routine returns the coefficients of a basis function for a given point in the unit triangle. Three basis functions are available:

<table>
<thead>
<tr>
<th>Basis</th>
<th>Coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppm_param_bem_basis_const</td>
<td>1</td>
</tr>
<tr>
<td>ppm_param_bem_basis_linear</td>
<td>3</td>
</tr>
<tr>
<td>ppm_param_bem_basis_quad</td>
<td>6</td>
</tr>
</tbody>
</table>

The following table shows how the basis functions are defined. It shows at what point the current coefficient is 1 and all others 0. $s$ and $t$ are the coordinates of the unit triangle with $s, t \in [0, 1]$ and $1 - s - t \geq 0$

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>s</th>
<th>t</th>
</tr>
</thead>
<tbody>
<tr>
<td>ppm_param_bem_basis_const</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>ppm_param_bem_basis_linear</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>ppm_param_bem_basis_quad</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

SUBROUTINE ppm_template_comp_boundary(id, xp, vp, Npart, &
                                     cd, ldc, Ncon, &
                                     fp, params, info)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id( :)</td>
<td>I</td>
<td>Local to global particle number mapping</td>
</tr>
<tr>
<td>xp(:, :)</td>
<td>F</td>
<td>The particle coordinates</td>
</tr>
<tr>
<td>vp(:, :)</td>
<td>F</td>
<td>Additional particle data</td>
</tr>
<tr>
<td>Npart</td>
<td>I</td>
<td>The number of particles</td>
</tr>
<tr>
<td>cd(:, :)</td>
<td>I</td>
<td>The connections</td>
</tr>
<tr>
<td>ldc</td>
<td>I</td>
<td>The length of the connections</td>
</tr>
<tr>
<td>Ncon</td>
<td>I</td>
<td>The number of connections</td>
</tr>
<tr>
<td>fp(:, :)</td>
<td>F</td>
<td>Change of particle data</td>
</tr>
<tr>
<td>params( :)</td>
<td>F</td>
<td>User defined parameter</td>
</tr>
<tr>
<td>info</td>
<td>I</td>
<td>Return status</td>
</tr>
</tbody>
</table>

This template is for carrying out integrations over triangles defined by connections of three particles. It gets called in the user’s program. The parameter
list is not fixed and should be modified by the user according to what has to be calculated or what is needed for the calculation. First all particles that are in the local connections have to be gathered. The gathering of the particles and their additional data is the same procedure as in the connection interaction template. Then a quadrature rule is set up and the basis function coefficients are calculated. Then the boundary integral of every local particle with every boundary element (triangle defined by a connection) is calculated. In order to get all interaction the local particle are send around the ring. This way every particle ‘sees’ every boundary element. Inside this template are a lot of comments which explain what is going on and hints where to modify or to add some code in order to customize this template.
3 CHARMS

The abbreviation CHARMS stands for "Conforming Hierarchical Adaptive Refinement Methods". In the field of simulation applications the finite element method is widely used. There, adaptive finite element solvers are of great use in order to reduce computational costs since they concentrate the computational power on the regions of interest. But it can be very hard to implement mesh refinement algorithms where the main problem is to maintain the integrity of the mesh (e.g. to avoid hanging nodes). For example in two dimensions with a triangular mesh it is still relatively simple to handle that (red, green, blue refinement), but in higher dimensions this can cause severe headache.

The approach of CHARMS is not to refine the elements but the basis functions. For that the basis functions have to be refinable which means that such a function can be expressed as a linear combination of finer basis functions [5]. Based on an initial mesh a hierarchy of nested self similar meshes are generated where nodes from a lower level are also nodes of all finer levels. This ensures compatibility between the levels and the meshes with its set of basis functions on each level are consistent.

The project described in this section is an implementation of the CHARMS framework as introduced by Eitan Grinspun et al. in [6] and with additional inspiration by [7]. It is a library written in C that provides the facilities for handling the geometry (including the generation of the hierarchical refinement), the basis functions defined on each level of the hierarchy and the (un)refinement of the basis functions. In the following section the usage of this library is explained and some sample applications are demonstrated.

3.1 Using CHARMS

In order to use the CHARMS API you have to include the header files that contain the data structure definitions and the prototypes of the API functions. This will look like

```c
#include "path/to/charms/include/charms.h"
#include "path/to/charms/include/functions.h"
```

The main data structure is the CHARMS structure, that holds the information about the geometry and its refinements, the boundary conditions, the basis functions with its segments and other variables that refer to the current state of the structure. The first step is to create a new instance of a CHARMS structure.

```c
CHARMS *c;
c = CHARMS_Init(1);
```

First an unassociated pointer to a CHARMS structure has to be defined. The call to the CHARMS_Init(...) function then allocates memory for the structure and returns a pointer to it. This pointer has to be passed as first parameter to every API function. This way more than one instance of a CHARMS structure can be used in parallel without generating any conflicts. The parameter (in the above example '1') denotes the dimensionality of the problem to be computed. In this case it will be a one dimensional problem. Dimensions up to 3 are supported.
The dimension also determines what type of computational elements may be used. The different element types are explained later in the geometry section.

After the use of a CHARMS structure one has to release the allocated memory in order to avoid massive memory leaks. Depending on the geometry and the level of refinement and the available memory a CHARMS structure can consume several hundred megabytes up to gigabytes. The following call to

```
CHARMS_Destroy(c);
```

frees all the used memory and the address c is pointing to is then no longer valid.

**3.1.1 Geometry data**

The geometry of the computational domain, respectively the computational elements the domain is composed of has to be added to the CHARMS structure. The geometry to be added consists of nodes and elements. The three (two) functions

```
CHARMS_AddNode(...)
CHARMS_AddNodes(...)
CHARMS_AddElement(...)
```

are for that purpose. Based on this geometry information the requested refinement levels and the basis functions on each of these levels will be generated. Before the elements can be added the corresponding nodes must be present since the elements are defined by the node numbers and not by the coordinates itself. To add 11 equidistant nodes between 0 and 1 (to stay with the one dimensional example above) the code for that would look like this:

```
for(i = 0; i <= 10; i++)
    data[i] = (real)i * 0.1;
CHARMS_AddNodes(c, 11, data);
```

This fills the array data with the coordinates of the nodes and adds them to the CHARMS structure c. The numbering of the nodes starts with 0. As soon as all nodes has been added the elements can be defined.

There are four types of elements available where the triangle can be used in 2D and 3D. The element types are shown in figure 4. The numbers at the nodes denotes the order they define the element. For both the triangle and the square element the right hand rule applies. The tetrahedron is defined such that the normal of the triangle defined by the nodes 0 to 2 points in direction of node 3. The CHARMS_LINE element can only be used in 1D. CHARMS_TRIANGLE and CHARMS_SQUARE both in 2D where they can be mixed. There is a restriction to the square element that it must be a parallelogram (in contrast to an arbitrary quadrangle). The triangle can also be used in 3D to represent e.g. a triangulated surface. The CHARMS_TETRAHEDRON is limited to 3D.

Going back to the example the elements can now be added. For this an array containing the indices of the nodes that define an element has to be passed to the CHARMS_AddElement(...) function:
\begin{figure}
\centering
\includegraphics[width=0.5\textwidth]{tetrahedron.png}
\caption{The four element types}
\end{figure}

```
j = 0;
for(i = 0; i < 10; i++) {
    e[0] = j++;
    e[1] = j;
    CHARMS_AddElement(c, CHARMS_LINE, e, -1, NULL, 0);
}
```

The parameter \(-1\) means that the element to be added does not have a parent element and \texttt{NULL} means that the element does not have any child elements. The last parameter is the refinement level the element belongs to. In this case it is the coarsest level. All other finer levels have a number greater than 0. With this function it is also possible to add an already refined geometry but then you have to take care of the parent-child relationship by yourself.

Now the geometry is defined. The computational domain is divided into 10 elements of the same length.

\subsection{Boundary conditions}

To add boundary conditions to a problem the affected entities of the geometry have to be marked with a user-defined number (identifier). With this number the boundary condition can be recognized as such during the calculation and the appropriate condition can be applied to it. The entities are nodes, edges and faces denoted by the parameters \texttt{CHARMS\_NODE}, \texttt{CHARMS\_EDGE} and \texttt{CHARMS\_FACE}. For every entity that has to be marked, a list of nodes that define it has to be passed to the \texttt{CHARMS\_AddBoundary(...) function} (the third parameter is the length of the list, respectively the number of nodes). In the example on both ends of the domain conditions will be defined:

```
e[0] = 0;
```
CHARMS_AddBoundary(c, CHARMS_NODE, i, e, 1);

e[0] = 10;
CHARMS_AddBoundary(c, CHARMS_NODE, i, e, 2);

On the left end applies boundary condition 1 and on the right end boundary condition 2. What these conditions mean depend on the user’s intent, e.g., 1 stands for a Dirichlet boundary and 2 for a Neumann boundary.

3.1.3 Refinement levels

After the nodes, elements and boundary conditions are defined, the geometry has to be refined and the basis functions has to be build over the elements:

CHARMS_BuildRefinement(c, 3);

This refines the geometry to 3 levels (0 to 2). If this value is set to 1 then no refinement will take place. The refinement of an element will lead to smaller and congruent elements of the same type. Edges are bisected, triangles and squares are quadrisection and the tetrahedron gets octisection. The boundary conditions are carried along the refinement, i.e., if an edge of an element has a condition defined on it, the corresponding edges of the children elements will carry the same boundary condition. As soon as the refinement is done, the basis functions get defined over the elements on every level. Each basis function is represented as a set of basis segments. A basis segment is that part of a basis function that is restricted to a specific element. Later in the calculation these basis segments are the objects that will be calculated with since the main iteration goes over the active elements. Therefore every active element carries a list of basis segments that are on the same level or on coarser (ancestor) level and every basis segment carries as list of boundary conditions that are part of this segment. Initially only the elements and basis functions of the coarsest level are active.

3.1.4 Calculation

The calculation of the left hand and right hand side of the system to solve is done with an iteration over every active element and over every combination of the same and ancestor level basis segments:

1: CHARMS_Element *e;
2: CHARMS_BasisSegment *s1, *s2;
3: CHARMS_BuildIntegrationElements(c);
4: for(i = 0; i < c->nelements; i++) {
5:     if(c->elements[i]->active == 0)
6:         continue;
7:     e = c->elements[i];
8:     for(j = 0; j < e->nsame; j++) {

22
```c
9:   s1 = e->same[j];
10:  lhs[s1->locid][s1->locid] += integrate(s1, s1);
11:  rhs[s1->locid] += load(s1);
12:  for(k = (j + 1); k < e->nsame; k++) {
13:      s2 = e->same[k];
14:      lhs[s1->locid][s2->locid] += integrate(s1, s2);
15:      lhs[s2->locid][s1->locid] += integrate(s2, s1);
16:  }
17:  for(k = 0; k < e->nancestor; k++) {
18:      s2 = e->ancestor[k];
19:      lhs[s1->locid][s2->locid] += integrate(s1, s2);
20:      lhs[s2->locid][s1->locid] += integrate(s2, s1);
21:  }
22:  for(k = 0; k < s1->nboundaries; k++) {
23:      if(s1->boundaries[k]->boundarytype == 1)
24:          ...  
25:      if(s1->boundaries[k]->boundarytype == 2)
26:          ...  
27:  }
28: }
29: }
```

First the integration elements have to be build (line 3). This means that for each active element the lists for the same level and ancestor level basis segments get compiled. This function has to be called at least once after the refinement levels have been build or if some basis functions have been activated or deactivated. After doing so the iteration can be performed.

Here is the point where it’s getting dirty. In order to iterate over every active element you have to touch some fields of the CHARMS structure directly. c->nelements contains the number of the stored elements either active or inactive. The elements itself are stored in the c->elements array that is an array of pointers to the elements. To test if an element is active the field active in the CHARMS_Element structure has to be 1. If not, the current element can be skipped (lines 5 and 6). Lines 8 to 28 illustrate the calculation of the left hand and right hand side. For the right hand side only the basis functions of the same level as the current element are considered. For the left hand side all combinations of the same level bases with itself and with the ones of the ancestor levels have to be computed. The number of basis segments on the same level is in e->nsame and the pointer to segments in the e->same array. For the ancestor segments its e->nancestor and e->ancestor. To store the results the array for the left hand and right hand side needs as much entries as there are active basis functions. The number of active basis functions is stored in c->active_basins.

Every segment then has a so called local ID which is a value between 0 and the number of active basis function exclusive. If a basis consists of 6 segments each
segment has the same local ID as the basis. So all integrations with one of these segments will contribute to the same basis.

For evaluating the integral there are the two functions

```c
CHARMS_HatFunction(...)  
CHARMS_DHatFunction(...)  
```

that will return the value of the linear basis function (in the case of quadrangle elements, the bilinear basis function) and its derivative over the given segment at a given point. The nodes of the element the segment is defined over are directly accessible with the segment. `s1->nnodes` contains the number of nodes (correspond to the number of nodes that define the current element) and `s1->nodes` are pointers to the nodes. The coordinates are then in the data array of a node, so the first coordinate of the first node of the segment that is `s1` is pointing to is `s1->nodes[0]->data[0]`. Please refer to the API reference or the provided demonstration applications to get a more detailed explanation of the use of these functions.

The number of boundary conditions per segment is stored in `s1->nboundaries`. The pointers to the boundary structures are stored in the `s1->boundaries` arrays. The user defined boundary identifier is stored in the field `boundarytype` of the boundary structure. Then the type of the first boundary condition (if present) is `s1->boundary[0]->boundarytype` (see lines 23 and 25). Depending on what the current boundary type represents, the according conditions have to be imposed.

After the left hand and the right hand side are computed the system of linear equations has to be solved. There the CHARMS library does not prescribe a special function for this and it is up to the user to apply an appropriate solver (the way the matrix and the load vector has to be stored is not prescribed neither). The file `tools.c` in the `examples/` directory contains such functions like an iterative Jacobi solver or a Gauss elimination algorithm as well as some quadrature rules for the different elements. But all functions in this file are not CHARMS specific and can be replaced by others.

The solution of the linear system contains the weights of the active basis functions. In order to get the solution at the nodes these weights have to be set

```c
CHARMS_SetWeights(c, weights);  
```

Then the solution at all nodes stored in the CHARMS structure can be gathered

```c
for(i = 0; i < c->nnodes; i++)  
  solution[i] = CHARMS_GetSolutionAtNode(c, i);  
```

or only the solution at the active nodes (the nodes that are center nodes of an active basis function)

```c
for(i = 0; i < c->nbasis; i++) {  
  if(c->basis[i]->active == 0)  
    continue;  
  
  solution[c->basis[i]->locid] = \n    CHARMS_GetSolutionAtNode(c, c->basis[i]->center);  
}  
```
3.1.5 Basis (un)refinement

Based on the solution an error estimator can be applied in order to refine or unrefine certain basis functions. The (un)refinement can only be applied to functions that are active. The exact criteria when a basis can be (un)refined can be found in the API specifications. In the worst case, if a basis cannot be (un)refined nothing happens.

CHARMS_RefineBasis(c, basisid);
CHARMS_UnrefineBasis(c, basisid);

The provided basis ID is the global ID of the basis as stored in c->basis[i]->id and correspond always to the index in the basis array in the CHARMS structure. With some bases (un)refined the computation can be redone in order to get more appropriate results.

In the examples/ directory are example implementations of a Poisson solver in 1D and 2D. In the 2D case it is also demonstrated how to mix triangle and rectangle elements. Another two applications solve the heat equation in 2D and on a triangulated surface in 3D.

3.2 Applications

In this section I present some applications that demonstrate the use of the CHARMS library. All applications can be found in the examples/ directory. The used helper functions for solving linear equations and the quadrature rules for the numerical integrations are in the tools.c file. The first application is a solver for the Poisson equation in 1D. The second one is for solving the Poisson equation in 2D. There the use of two different element types (triangles and squares is also demonstrated). Then comes a solver for the diffusion equation in 2D and 3D. 2.5D means that the heat equation is solved on a triangulated surface in 3D. For displaying the results of the examples their output is in Matlab syntax.

3.2.1 1D Poisson equation

Consider the following problem: Find a function $u$ such that

$$-u''(x) = f(x) \quad x \in \Omega$$

with the boundary conditions $u(1) = h$ and $-u'(0) = g$. The application of the Galerkin approximation method with a set of $N$ linear basis functions $\eta_i(x), i = 1, \ldots, N$ leads to a system of $N - 1$ linear equations of the form

$$Ax = b$$

with

$$A_{ij} = \int_{\Omega} \nabla \eta_i(x) \nabla \eta_j(x) dx$$

$$b_i = \int_{\Omega} \eta_i(x) f(x) dx + g \eta_i(0) - h \int_{\Omega} \nabla \eta_i(x) \nabla \eta_N(x) dx$$

25
Figure 5: Exact solution of the Poisson problem

For the current example the computational domain is $\Omega = [0, 1]$ that is subdivided into 10 elements, therefore is $N = 11$. In order to have an exact solution to compare the calculated results with, $u$ is predefined as

$$u(x) = 100x^5 - 300x^4 + 325x^3 - 150x^2 + 25x$$

and the function $f$ and the values for $g$ and $h$ can be derived from it

$$f(x) = -2000x^3 + 3600x^2 - 1950x + 300$$

$$u(1) = h = 0$$

$$-u'(0) = g = -25$$

The following plots (figure 6) shows the initial mesh of the unweighted basis functions, the weighted basis functions and the Galerkin solution. The weights of the basis functions are the result of applying the method to the above values for $f$, $g$ and $h$. In order to get the Galerkin solution the contributions of all weighted basis function have to be added up over each element. This leads to a piece wise linear approximation of the exact solution.

This approximation can now be improved by refining some basis functions where the Galerkin solution deviate too much from the exact solution. If an exact solution is not available an a posteriori error estimator based on the second derivative of the numerical solution compared to $f$ can be used. Such a refined mesh of basis functions is shown in the lower half of figure 6. The refinement resulted in 23 basis functions on 3 levels where some elements are one fourth of the size of an original element.

Figure 7 shows the the basis functions and the Galerkin solution with 23 bases but this time the 23 basis functions are all on the same level with 22 equally sized elements. Compared to the adaptively refined mesh, this gives less accurate results (especially in the interval $[0, 0.3]$) with the same number of bases. On the other hand the refined mesh has 34 active integration elements in contrast to only 22 active elements in the unrefined case. With the refined mesh the time to calculate the entries of the stiffness matrix and the load vector takes a little bit more time but the systems of linear equations to solve are of the same size in both cases and the results of the refined mesh are more accurate.
Figure 6: Upper three plots: Unweighted and weighted set of basis function with no refinement (1 level) and the resulting Galerkin solution. Lower three plots: Unweighted and weighted set of basis function with two refinements (3 levels) and the resulting Galerkin solution.
Figure 7: Unweighted and weighted set of 23 basis function with no refinements and the resulting Galerkin solution.

### 3.2.2 2D Poisson equation

The Poisson problem in 2D on a polygonal domain $\Omega$ with boundary $\Gamma$ is

\[
- \Delta u = f \quad \text{in } \Omega \\
\quad u = u_D \quad \text{on } \Gamma_D \\
\quad \frac{\partial u}{\partial n} = g \quad \text{on } \Gamma_N
\]

with $\Gamma_D$ a closed subset of $\Gamma$ of positive length with Dirichlet boundary conditions. The remaining part $\Gamma_N := \Gamma \setminus \Gamma_D$ has Neumann boundary conditions. In this example $\Gamma_D$ is the left and upper side (outer rim) and the ‘curved’ part (inner rim) of the boundary. $u_D$ is 0. $\Gamma_N$ then is at the bottom and the right side of the boundary and $g$ is 0 too (see figure 8). $f$ is constant 1 on the whole domain. The domain is triangulated with 15 nodes (of which are 10 Dirichlet nodes) and 16 triangles. The resulting Galerkin discretization has a similar form as in the one dimensional case.

The refinement criterion for a basis function is the same as in the one dimensional case. The maximum number of refinement levels is restricted to 4. The following table lists the number of active basis functions and active integration elements after each refinement step. Due to the refinement scheme the number of active basis functions for example of level 2 contains also the one of level 0 and 1. The same applies to the active elements. That means that the third refinement introduced 48 new active basis functions and 136 new active elements.
Figure 8: Left: Triangulated computational domain. Right: Mixed triangular and square elements

<table>
<thead>
<tr>
<th>Refinement</th>
<th>Active bases</th>
<th>Active elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>23</td>
<td>40</td>
</tr>
<tr>
<td>3</td>
<td>71</td>
<td>176</td>
</tr>
<tr>
<td>4</td>
<td>269</td>
<td>720</td>
</tr>
</tbody>
</table>

The following plots (figure 9) show the solution on the different refinement levels. The plotted meshes show the active integration cells. These cells are the smallest elements that support a (refined) active basis function.

CHARMS can also be applied to mixed element configurations. Here, the same domain with the same boundary conditions is chosen but this time decomposed into triangles and squares. Only the solution on the initial mesh and after 3 refinement steps are shown (figure 10).

3.2.3 2D Heat equation

To solve the heat equation

$$\frac{\partial u}{\partial t} = \Delta u + f \text{ in } \Omega \times [0,T]$$

numerically, the implicit Euler scheme in time is applied where the interval $[0,T]$ is split into $M$ equally sized intervals of length $dt = T/M$ and this results in the equation

$$(id - dt\Delta)u_m = df_m + u_{m-1}$$

where $f_m = f(x,t_m)$ and $u_m$ is the approximation of $u$ at time $t_m = mdt$. The Galerkin method applied at every time step leads to a system of linear equations of the form

$$(dtA + B)U_m = dtb + BU_{m-1}$$

where the stiffness matrix $A$ and the load vector $b$ have the same form as in the two dimensional Poisson equation. The mass matrix $B$ is

$$B_{ij} = \int_{\Omega} \eta_i(x)\eta_j(x)dx$$

The computational domain is the one shown on the left of figure 8. The Dirichlet conditions on the outer rim are $u_D \equiv 1$ and on the inner rim $u_D \equiv 0$. 

29
Figure 9: Galerkin solution of the Poisson equation in 2D with refinement levels 1-4. Left-right, top-down: 1 (no refinement), 2, 3 and 4 level refinement.

Figure 10: Galerkin solution of the Poisson equation in 2D with mixed elements. Left: no refinement (1 level). Right: 4 level refinement.

The Neumann conditions are $g \equiv 0$ and $f \equiv 0$. The time step is $dt = 0.01$. Figures 11 and 12 show the refinement of the initial mesh at the first time step. Only these functions were refined in areas with rapid changes. Figures 13 shows the time evolution of the system at times $T = 0.25, 0.5$ and $1.0$.

3.2.4 Heat equation on a sphere

This example has the exact same formulation as in the two dimensional case. The only difference is that the elements are no longer in 2D but embedded into
Figure 11: Galerkin solution of the heat equation in 2D at $T = 0.01$ with refinement levels 1-4. Left-right, top-down: 1 (no refinement), 2, 3 and 4 level refinement.

3D. The computational domain is now the triangulated surface of a sphere [8]. To evaluate a basis function over a triangle arbitrarily positioned in space it is transformed such that it is parallel to the $(x,y)$-plane. Then the basis function can be evaluated as in 2D. Here, $f$ and $g$ are both 0 and one nodal Dirichlet condition is given. This sets the highest node $(0,0,1)$ to 1 such that it acts as a permanent heat source that will finally heat up the whole surface of the sphere.

This time only three refinement levels are allowed which concentrates around the heat source. The time step is $dt = 0.01$. Figure 14 shows the refinement of the initial mesh up to two levels. The third refinement step is shown in figure 15. Figure 16 demonstrates the time evolution of the system at times $T = 0.5, 1.0, 5.0$ and 10.0. At $T = 10.0$ the sphere got almost heaten up completely.
Figure 12: Galerkin solution of the heat equation in 2D at $T = 0.01$ with refinement (5 level)

Figure 13: Time evolution of the heat equation in 2D at $T = 0.25, 0.5$ and $1.0$
Figure 14: Galerkin solution of the heat equation on a surface in 3D at $T = 0.01$ with refinement levels 1 and 2.
Figure 15: Galerkin solution of the heat equation on a surface in 3D at $T = 0.01$ with 3 refinements.

Figure 16: Time evolution of the heat equation on a surface in 3D at $T = 0.50, 1.0, 5.0$ and 10.0.
4 Conclusions

With the work on the PPM project I had a very interesting view into the development of a modern high performance multiprocesssing library for particle simulations. My contributions to this library are the ring topology, the connection handling and some helper routines for calculating boundary integrals. The ring topology can be used to do calculation where every particle in the system has to interact with each other, or just to send arbitrary data around in order to distribute them among the processors. The connection handling is for keeping user defined relations between particles consistent while the particles are distributed to different processors. Then the helper routines consists of a template for calculating boundary integrals which makes use of the ring topology and the connections handling. The other routines are quadrature rules and interpolation schemes for triangles, the boundary elements that are defined by connections. Last but not least I had my first close encounter with the FORTRAN programming language with the work on the PPM library.

The second project was the implementation of the CHARMS framework as suggested by E. Grinspun et al. It is a new method of adaptively enhancing the finite element solution space based on the refinement of the basis functions and not the elements. While implementing it I gained new skills in the finite element method and how to represent the geometry and the basis functions with its levels of refinement in the C programming language.

5 Outlook

For the PPM part the place where the ring topology and the connections come together is for molecular dynamics simulations or the calculation of boundary integrals where e.g. three connected particles represent molecules or boundary elements. The provided template for calculating the boundary integrals is not tested yet and there is still some work to do to make a boundary integral solver out of it.

The CHARMS framework still offers some points of improvements. These are for example a more general solver framework that makes use of the CHARMS library similar to the ones in the example applications. Within this solver framework a need for more sophisticated error estimators and (un)refinement criteria is present. Another point is to implement more elements and more basis functions other than only linear hat functions.
6 References


A  CHARMS API Reference

A.1  Initializing

CHARMS *CHARMS_Init(int dim);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>The number of spacial dimensions</td>
</tr>
</tbody>
</table>

This function creates a new CHARMS structure for a problem in dim dimensions. dim may have values from 1 to 3. Some memory for the data structures that represent the geometry and the basis functions will be pre-allocated. The return value is a pointer to the allocated CHARMS structure or NULL if something went wrong.

A.2  Destroying

void CHARMS_Destroy(CHARMS *c);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
</tbody>
</table>

This function deallocates the memory of all used data structures and finally deallocates the CHARMS structure itself. After a call to this function the given pointer to the CHARMS structure is no longer valid and the use of it will lead to segmentation faults. No return value.

A.3  Adding nodes

int CHARMS_AddNode(CHARMS *c, real *data);
int CHARMS_AddNodes(CHARMS *c, int n, real *data);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
<tr>
<td>n</td>
<td>Number of nodes to add</td>
</tr>
<tr>
<td>*data</td>
<td>Array of coordinates of the node(s)</td>
</tr>
</tbody>
</table>

These two functions add nodes to the geometry. data is an array holding the coordinates of the node(s). In the case of the first function only one node will be added and the data array should be as long as the dimensionality of the given CHARMS structure. The return value is the ID of the added node or -1 if the node could not be added. The ID of the first node is 0. The other function adds n nodes to the CHARMS structure therefore the data array should be long enough to hold at least that much nodes. The return value is the number of added nodes or 0 if something went wrong.

A.4  Adding elements

int CHARMS_AddElement(CHARMS *c, int type, int *nodes,
                                   int parent, int *children, int level);
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
<tr>
<td>type</td>
<td>The element type</td>
</tr>
<tr>
<td>*nodes</td>
<td>Array of node IDs</td>
</tr>
<tr>
<td>parent</td>
<td>The ID of the parent element</td>
</tr>
<tr>
<td>*children</td>
<td>The IDs of the child elements</td>
</tr>
<tr>
<td>level</td>
<td>The refinement level this element is in</td>
</tr>
</tbody>
</table>

This function adds an element to the geometry. The type of an element is given by the following defines. The number of nodes per element and the number of child elements is given by the type.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>#Nodes</th>
<th>#Children</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARMS_LINE</td>
<td>A line in 1D</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>CHARMS_TRIANGLE</td>
<td>A triangle in 2D or 3D</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>CHARMS_SQUARE</td>
<td>A rectangle in 2D</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>CHARMS_TETRAHEDRON</td>
<td>A tetrahedron in 3D</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

parent is the ID of the parent element or -1 if an element does not have a parent. The children array should have as many entries as mentioned in the table above. If an element does not have any children, the NULL pointer should be given. level indicates the refinement level this element is in where a value of 0 denotes the coarsest level. To define a geometry on only the coarsest level parent is always -1, children is always NULL and level is always 0. All elements of level 0 will be active by default. The return value is the ID of the currently added element. The ID of the first element is 0.

A.5 Adding boundary conditions

int CHARMS_AddBoundary(CHARMS *c, int type, int nnodes, int *nodes, int boundary);

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
<tr>
<td>type</td>
<td>The boundary condition type</td>
</tr>
<tr>
<td>nnodes</td>
<td>Number of nodes of the boundary</td>
</tr>
<tr>
<td>*nodes</td>
<td>Array of node IDs</td>
</tr>
<tr>
<td>boundary</td>
<td>The boundary condition identifier</td>
</tr>
</tbody>
</table>

This function adds boundary condition identifier to the CHARMS structure. With this you can assign a number to a geometric object such as a node, edge or face. The defined types are given below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARMS_NODE</td>
<td>Node boundary condition</td>
</tr>
<tr>
<td>CHARMS_EDGE</td>
<td>Edge boundary condition</td>
</tr>
<tr>
<td>CHARMS_FACE</td>
<td>Face boundary condition</td>
</tr>
</tbody>
</table>

These boundary condition identifiers does not have an influence of the geometry but they need to be known in order to carry them along with the refinement.
Later in the integration loop the boundary can be recognized by its given identifier and the appropriate condition can be applied to them. The number of nodes and the list of node IDs (\texttt{nodes and node}) correspond to the geometric object this boundary condition should be applied to. Let assume that an edge of a triangle (\texttt{CHARMS\_TRIANGLE element) is the boundary of a domain. Then the type is \texttt{CHARMS\_EDGE} and \texttt{nodes} has the value 2 and \texttt{nodes} has two entries with the node IDs in it that define the edge. The value for \texttt{boundary} can be anything. The return value is the ID of the boundary condition or -1 if something went wrong.

A.6  Building the refinement levels

\begin{verbatim}
int CHARMS_BuildRefinement(CHARMS *c, int nlevel);
\end{verbatim}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
<tr>
<td>nlevel</td>
<td>Number of nodes of the boundary</td>
</tr>
</tbody>
</table>

This function builds the geometric refinement and finally associates the basis functions to the nodes and elements on the different levels. All basis function of level 0 will be active by default. \texttt{nlevel} determines the number of levels that have to be created. A value of 1 means that no refinement will take place. The refinement will only be done if there are only elements of level 0 in the geometry, otherwise it is assumed that an already refined geometry has been added to the CHARMS structure. The return value is 1 on success or 0 if something went wrong.

A.7  Building the integration elements

\begin{verbatim}
int CHARMS_BuildIntegrationElements(CHARMS *c);
\end{verbatim}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
</tbody>
</table>

This function prepares every active element for the integration loop. That means that it iterates over every active element and assigns the same level and ancestor level basis function segments (basis segments are these parts of a basis function that are defined over a certain element) to it and reset the weights of every active basis function to 1. The return value is 1 on success or 0 if something went wrong.

A.8  Refining a basis

\begin{verbatim}
void CHARMS_RefineBasis(CHARMS *c, int basis);
\end{verbatim}

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
<tr>
<td>basis</td>
<td>The ID of the basis to be refined</td>
</tr>
</tbody>
</table>

This function refines the basis with the ID given in \texttt{basis}. The ID is the global ID of a basis function. Only already active basis functions can be refined.
If the given basis function is not active nothing will happen. All odd and not already active children of the basis function gets activated and their weights are set to 0. The elements that support the activated children gets also activated. No return value.

A.9 Unrefining a basis

```c
void CHARMS_UnrefineBasis(CHARMS *c, int basis);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
<tr>
<td>basis</td>
<td>The ID of the basis to be unrefined</td>
</tr>
</tbody>
</table>

This function unrefines the basis with the ID given in `basis`. The ID is the global ID of a basis function. A basis function may only be unrefined if it is active and none of its children is refined. Otherwise nothing will happen. Then all odd and active children gets deactivated. The supporting elements of these deactivated basis functions gets deactivated only if there is no other active basis function that is supported by this element. No return value.

A.10 Setting the weights

```c
int CHARMS_SetWeights(CHARMS *c, real *weights);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
<tr>
<td>*weights</td>
<td>Array of the new weights</td>
</tr>
</tbody>
</table>

This function sets the weights of every active basis function. The array with the weights has to be as long as there are active basis function in the current setup. The first entry in the array is the weight for the basis function with the local ID of 0 and so on. All other weights (i.e. the weights of the not active functions) are reset to 1. The return value is 1 on success or 0 if something went wrong.

A.11 Getting the solution at a node

```c
real CHARMS_GetSolutionAtNode(CHARMS *c, int nodeid);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
<tr>
<td>nodeid</td>
<td>The ID of a node</td>
</tr>
</tbody>
</table>

This function returns the solution at the node with the given ID. The solution is calculated out of the weighted basis functions that have this node in their support.
A.12 Setting the weights and getting the solution

```
int CHARMS_GetSolution(CHARMS *c, real *weights, real *solution);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
<tr>
<td>*weights</td>
<td>Array of the new weights</td>
</tr>
<tr>
<td>*solution</td>
<td>Array for the solution</td>
</tr>
</tbody>
</table>

This function is a wrapper for the previous two functions for setting the weights and getting the solution at a specific node. The weights are set with CHARMS_SetWeights(...) and the solution array is filled only with the solution at the nodes that are the center nodes of the active basis functions. The weights and solution arrays have to be of the same length. The return value is 1 on success or 0 if something went wrong.

A.13 Verbose output

```
void CHARMS_Print(CHARMS *c);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*c</td>
<td>Pointer to a CHARMS structure</td>
</tr>
</tbody>
</table>

This function prints out the contents of the given CHARMS structure. The output is written to stdout. For complex geometries and/or for many refinement level this can become very much. No return value.

A.14 Evaluating basis functions

```
real CHARMS_HatFunction(CHARMS_BasisSegment *s, real *x);
void CHARMS_DHatFunction(CHARMS_BasisSegment *s, real *x,
                          real *res);
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*s</td>
<td>Pointer to a CHARMS_BasisSegment structure</td>
</tr>
<tr>
<td>*x</td>
<td>The point to evaluate at</td>
</tr>
<tr>
<td>*res</td>
<td>The result at x</td>
</tr>
</tbody>
</table>

These functions evaluates a hat function or its derivative over an element at the point given by x. This point has to be inside the element otherwise wrong results will be evaluated. The derivative returns the result in the res array. The hat functions and its derivatives are currently defined in the following configuration of elements types and spacial dimension

<table>
<thead>
<tr>
<th>Element type</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARMS_LINE</td>
<td>1D</td>
</tr>
<tr>
<td>CHARMS_TRIANGLE</td>
<td>2D, 3D</td>
</tr>
<tr>
<td>CHARMS_SQUARE</td>
<td>2D</td>
</tr>
<tr>
<td>CHARMS_TETRAHEDRON</td>
<td>3D</td>
</tr>
</tbody>
</table>

If the element type and the spacial dimension do not correspond the return value will be 0 in the case of the hat function. In the case of the derivative, the res array will not be altered in any way.
B CHARMS Implementation details

The current CHARMS implementation is designed as a library that provides the facilities for handling the geometry and its refinement and the basis functions that are defined at every node and every refinement level. The source code of the library consists of the following files that can be found in the CHARMS directory:

- **Makefile**: Makefile for the library.
- **config.h**: Configuration file for compile-time parameter.
- **src/charms.c**: The CHARMS API and internal functions.
- **src/functions.c**: The API for the basis functions.

The header files to be included (charms.h and functions.h), containing the prototypes of the API and of the internal functions and the structures, are located in the include/ subdirectory.

B.1 Compiling

Change into the CHARMS directory and just type `make`. This will compile the CHARMS library into the lib/ subdirectory as `libcharms.a`. In the file `config.h` you can change the compile-time parameter that defines if single or double precision for floating point numbers should be used. After changing something in this file you have to recompile the library.

The subdirectory `examples/` contains the previously described example applications of the library. There, just type `make` again and all examples will be compiled. The executables can then be found in the corresponding subdirectory.

B.2 Naming conventions

To avoid conflicts with other source code or used libraries, every function, structure or parameter has the prefix `CHARMS`. In order to distinguish between an API function and an internal function (public respectively private functions, to use the nomenclature introduced with the object oriented programming), private functions have an additional underscore as prefix. With this convention a public function will look like `CHARMS_Foo(...)` and a private function will look like `CHARMS_Bar(...)`. Private functions are only for internal use and should not be called by the user of the library directly. The actual name of a function starts with a capital letter and is continued with lower case letters. If the name is composed of more than one word, every word begins with a capital letter (like `CHARMS_FooBar(...)`).

B.3 Data structures

In this section I describe the data structures and how they represent the geometry and the basis functions. The data structures used by the CHARMS library are defined in the include/charms.h header file and will be explained in this section in more detail. The implementation consists of the following eight data structures
typedef struct CHARMS {
    int dimension;
    int nlevel;
    int nnodes, nodepool;
    CHARMS_Node **nodes;
    int nedges, edgepool;
    CHARMS_Edge **edges;
    int nffaces, facepool;
    CHARMS_Face **faces;
    int nelements, elementpool;
    CHARMS_Element **elements;
    int nboundaries, boundarypool;
    CHARMS_Boundary **boundaries;
    int nbasis, basispool;
    CHARMS_Basis **basis;
    int nsegments, segmentpool;
    CHARMS_BasisSegment **segments;
    int active_basis;
    int active_elements;
} CHARMS;

For example the list of nodes is the array nodes (node pool) where nnodes holds the number of nodes. The pool has a certain amount of preallocated structures such that each request for a new one will not invoke a memory allocation routine. If the number of entries in a pool exceeds the current pool size (e.g. nodepool) then a new bunch of structures of the corresponding type will be preallocated. This scheme is also used for the remaining structures. Besides that, the structure holds the current state of a CHARMS instance like the number of refinement levels, dimensions, active elements and basis functions.

There are four structures to store the geometric informations that are nodes, edges, faces and elements. An element consists of the former three geometric entities.

typedef struct CHARMS_Node {
    int id;
    int ndata;
    real *data;
    int nelements, *elements;
}
int nedges, *edges;
int nfaces, *faces;
int nbasis, *basis;
int boundary;
} CHARMS_Node;

Every node has an unique ID (id). This ID corresponds to the index where this node is located in the node pool. ndata is the length of the array data, where the coordinates of this node are stored in, is pointing to. A node keeps track of what geometric element it is part of. So it stores of what elements (nelements, elements), edges (nedges, edges) and faces (nfaces, faces) it is part of. These arrays contain the indices of these structures in the corresponding pool. The basis array has as many entries as there are refinement levels (nbasis contains the number of refinement levels). If a node is the center of a basis in level \( k \) then the \( k^{th} \) entry of the array contains the index of that basis, otherwise it contains the value \(-1\). The boundary is either \(-1\) if no boundary condition is defined for this node or it contains the index of the boundary condition.

typedef struct CHARMS_Edge {
  int id;
  int type;
  int level;
  int nnodes, *nodes;
  int nelements, *elements;
  int nfaces, *faces;
  int nsplit, *split;
  int boundary;
  real area;
} CHARMS_Edge;

Every edge has an unique ID (id). This ID corresponds to the index where this edge is located in the edge pool. The type defines the kind of this edge, respectively of how many nodes it consists of. Currently only edges with two nodes are implemented and the type is CHARMS_EDGE2NODE. level stores the refinement level this edge belongs to. nnodes and nodes stores how many and what nodes (the indices of them) this edge consist of. nelements, elements, nfaces and faces correspond to the same purpose as in the node structure. split is an array holding the node IDs of the nodes that resulted in refining this edge. nsplit then holds the number of the nodes. This information is stored for the situation where an edge belongs to more than one element. For example the refinement (or split) of an edge with two nodes will result in a new node in between. Since the refinement of a mesh is done element wise where every edge in the element gets refined, it can happen that an edge gets refined more than once and that will result in nodes that have the same coordinates. To avoid this waste of memory, it is stored if an edge has already been split up and what node was the result of the splitting. The boundary is either \(-1\) if no boundary condition is defined for this edge or it contains the index of the boundary condition. area is the length of this edge.

typedef struct CHARMS_Face {

```c
int id;
int type;
int level;
int nnodes, *nodes;
int nedges, *edges;
int nelements, *elements;
int nsplit, *split;
int boundary;
real area;
}

typedef struct CHARMS_Element {
  int id;
  int type;
  int nnodes, *nodes;
  int nedges, *edges;
  int nfarms, *faces;
  int nbasis, *basis;
  int parent;
  int nchildren, *children;
  int level;
  int active;
  real area;
  int nsame, same_size;
  CHARMS_BasisSegment **same;
  int nancestor, ancestorsize;
  CHARMS_BasisSegment **ancestor;
} CHARMS_Element;
```

Every face has an unique ID (id). This ID corresponds to the index where this face is located in the face pool. The type defines the kind of this face, respectively of how many nodes it consists of. Currently two types are defined, CHARMS_FACE_TRIANGLE with three nodes and CHARMS_FACE_SQUARE with four nodes. nnodes, nodes, nedges and edges stores of how many nodes/edges and what nodes/edges (the indices of them) this face consist. nelements and elements correspond to the same purpose as in the node or edge structure. split is an array holding the node indices of the nodes that resulted in the refining of this face. nsplit then holds the number of the nodes. The boundary is either −1 if no boundary condition is defined for this face or it contains the index of the boundary condition. area is the area of this face.

Every element has an unique ID (id). This ID corresponds to the index where this element is located in the element pool. The type is one of the four element type as mentioned in the API reference. The three arrays nodes, edges and faces contains the indices of the entities it consists of. basis is an array of the basis functions that are spanned over this element and that are on the same level. parent is the index of the parent element where the value is −1 if this element does not a parent. The number of children (nchildren) is determined by the element type since an element gets only refined in elements of the same type. If an element does not have children, all entries in the children array are −1. level is the level the element is in where 0 is the coarsest level. active is 1 if
the element is active or 0 otherwise and the area of the element is stored in area. The remaining fields store the information about the basis segments on the same level and the ancestor levels. The segments are stored in arrays of pointers where nsame and ancsame are the current number of segments in the array and nsamesize and ancsamesize hold the actual size of these arrays. Since these arrays get reconstructed every time CHARMS_BuildIntegrationElements(...) gets called they will not be deallocated and then newly allocated. For example if the number of ancestor level segments gets bigger than ancsamesize, the ancestor array will be reallocated to fit the new size. Later, if ancestor gets smaller the array will not be resized. This saves time but needs only a little more memory since the arrays are only arrays of pointers.

typedef struct CHARMS_Boundary {
    int id;
    int type;
    int boundarytype;
    int modes;
    CHARMS_Node **nodes;
    int entity;
    int level;
} CHARMS_Boundary;

Every boundary has an unique ID (id). This ID corresponds to the index where this boundary is located in the boundary pool. The type denotes if the boundary is node, edge or face boundary as provided by the user. boundarytype is the identifier of the boundary condition. The nodes the boundary is defined of are stored in an array of pointers to the nodes. The entity is the index of the corresponding entity in the pool depending of the type. I.e. if the type is a CHARMS_EDGE then entity hold the index of the edge that boundary applies to. level contains the same value as stored in the level field in the entity. In the case of a node boundary level has a value of -1 since one node can be in more than one refinement level.

Then two structures are needed to store the information about the bases and its segments.

typedef struct CHARMS_Basis {
    int id, locid;
    int center;
    int odd;
    int active;
    int level;
    int refined;
    int nchildren, *children;
    int nsegments, *segments;
} CHARMS_Basis;

Every basis has an unique ID (id). This ID corresponds to the index where this basis is located in the basis pool. The local ID locid is assigned to all active basis functions and may change with every call to CHARMS_BuildIntegrationElements(...). center is the index of the center node of the basis function (where the unweighted function has the value 1). odd is 1 if the center node
of the basis is not the center node of a basis on a coarser level. active and level have the same purpose as in an element. If the basis has been refined it is denoted by a value of 1 of the field refined, 0 otherwise. The indices of the children bases are stored in the children array and the indices of the basis segments the basis consists of is in segments. nchildren and nsegments are the length of the corresponding arrays.

typedef struct CHARMS_BasisSegment {
  int id, locid;
  int support;
  int basis;
  int level;
  int nmodes;
  CHARMS_Node **nodes;
  int nboundaries;
  CHARMS_Boundary **boundaries;
  real weight;
  real area;
  int type;
} CHARMS_BasisSegment;

Every basis segment has an unique ID (id). This ID corresponds to the index where this basis segment is located in the segment pool. The local ID locid as the one associated to the basis the segments belongs to. support is the index of the element that supports the segment and therefore the nodes in the nodes array are the same as in the element. The level is the same as in the supporting element or the basis the segment belongs to. boundaries is an array of pointers to the boundary conditions (if existing). The weight of a basis function is stored in the segment, so each segment of a basis has the same weight. The area is the same as in the supporting element. The type identifies the segment such that the appropriate function can be called to evaluate the basis function at a given point inside this segment.