New Design PMacc Resource Register

"DataConnector on Steroids"

Axel, René, Alex

13.02.2017

1 Graph based Resource and Event Managment

At the moment a big problem of PIConGPU is the explicit and fault-prone event handling. Events are created and maintained by hand. The goal of this work is to described and implement a resource access scheduling solution, which is able to calculate dependency graphs out of resource access informations. The atomic element is a *Functor*, which works on *Resources*. The used resources and the kind (and area) of them are annotated to the Functor as input parameter for the dependency graph creation. There exist two graphs. First of all the dependency graph is created out of the resource access informations. With this a concrete scheduling graph is created, which defines the order and concurrency of functors. The creation of the dependency graph and the execution of the scheduling graph is done automatically, but the creation of the schedule graph offers many optimization possibilities.

2 Dependency Graph

This parts contains general thoughts, possibilities and constraints about building the dependency graph. The general idea is, that every functor without unresolved dependency is ready to get scheduled[1].

2.1 Code-Order == Order of Accesses in Time

The dependency graph itself is built implicit. In code a sequence of *functors* is defined, which defines an order between them. Furthermore every functor defines a resource list determining, e.g. whether this functor reads, writes, or maybe atomically writes the resource. This can be done for multiple resources, but for now we concentrate on one resource. These information can be used to automatically generate a dependency graph as seen in figure 1 on the following page.

[1] Although there will be cases shown in section 3, where a functor without remaining dependency must not start

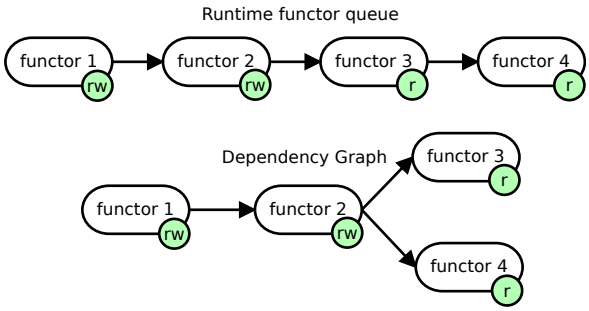


Figure 1: Code sequence of functors with resource access flags resulting in a dependency graph. A colored circle on a functor shows, whether a resource is accessed and how. In this example only one (green) resource is considered.

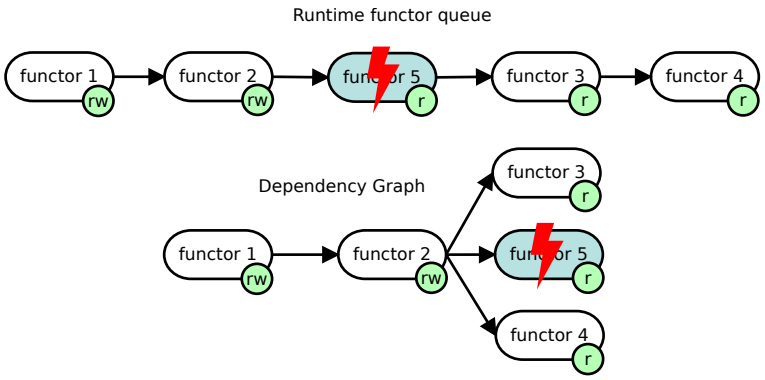


Figure 2: Addition of a new functor to an existing sequence is only allowed at the end.

2.2 Append only

As the dependency graph is created or changed as new functors are added, it shall not be possible to add new functors between already existing functors as seen in figure 2.

On the other hand it is one fundamental idea of this functor management, that a functor can be added at any time at runtime. A code could look like this (very simplified and only for one resource):

```
1 while (running)
2 {
3   add_functor(functor_1,"rw");
4   add_functor(functor_2,"rw");
5   add_functor(functor_3,"r");
6   add_functor(functor_4,"r");
7 }
```

This would create the already seen runtime functor queue and dependency graph in figure 1 for one loop run. While the second run the dependency graph will extent as seen in figure 3 on the next page.

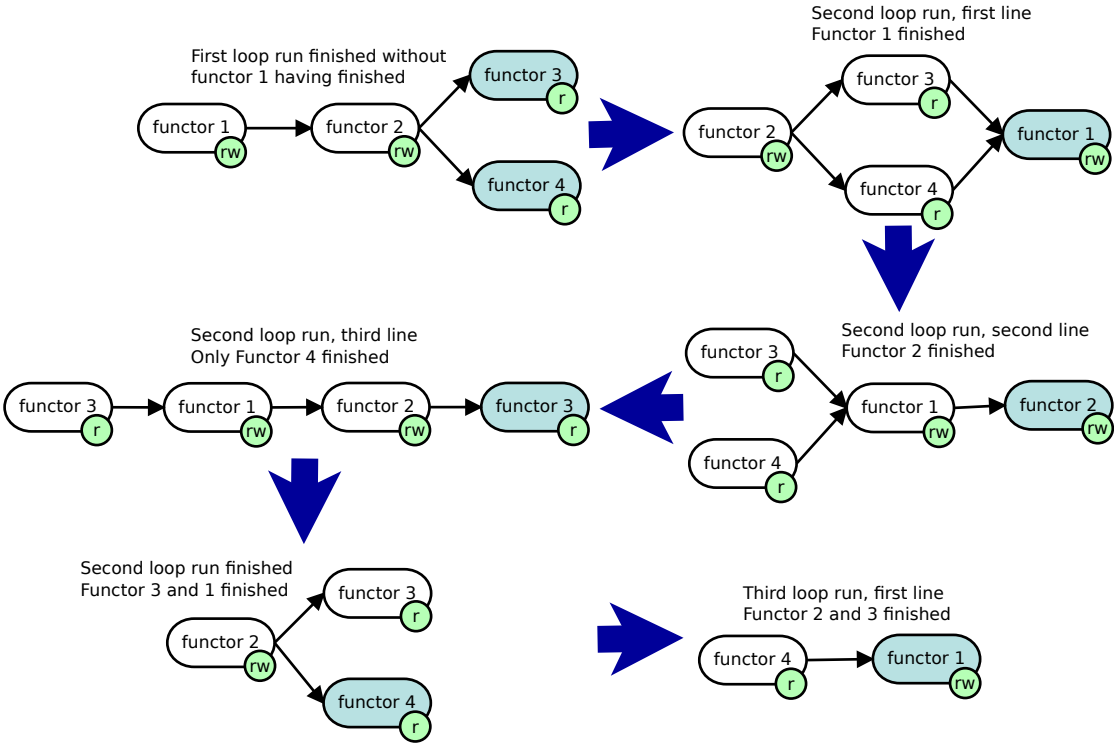


Figure 3: Adding and removing functors to and from the dependency graph run in parallel to the scheduling (and of course the creation of the scheduling graph) itself. Although the very same code is run in a loop, depending on the time behavior of the functors, different dependency graphs can occur.

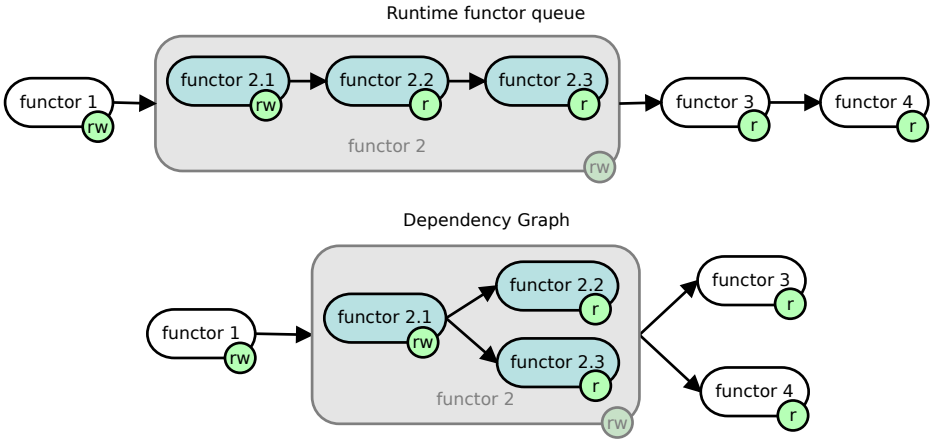


Figure 4: Existing functors can be refined, even at runtime, but they resulting graph must have no outer access and it must no promotion happen.

2.3 Splitting of functors

However it is possible to split an existing functor in multiple sub functors. With this, functor graphs can even be generated or refined at runtime as seen in figure 4. It is important to state, that no outer access of these sub functors is possible and the total access type is not allowed to get promoted (see also section 2.4), e.g. it is not allowed to refine a functor, which only reads from a resource, with a sub functor, which writes to a resource. The functor has its own sub dependency graph. If all sub functors are finished, the parent functor is finished.

2.4 Demotion of functors

At runtime it shall be possible to demote (but never to promote) the resource access flags (see also section 4.2 on page 8), which allows to rebuild a more parallel dependency graph at run time, e.g. if a read-write, which needs to be waited for by reads, demotes to a read or disappears at all as seen in figure 5 on the next page.

2.5 Multiple resources

In field more than one resource will be accessed by different functors, which all have their own access properties as seen in figure 6 on the following page.

3 Scheduling graph

The task of the scheduler is to create a scheduling graph out of the dependency graph. Thereby it is only allowed to add edges, but not to remove some as seen in figure 7 on page 6.

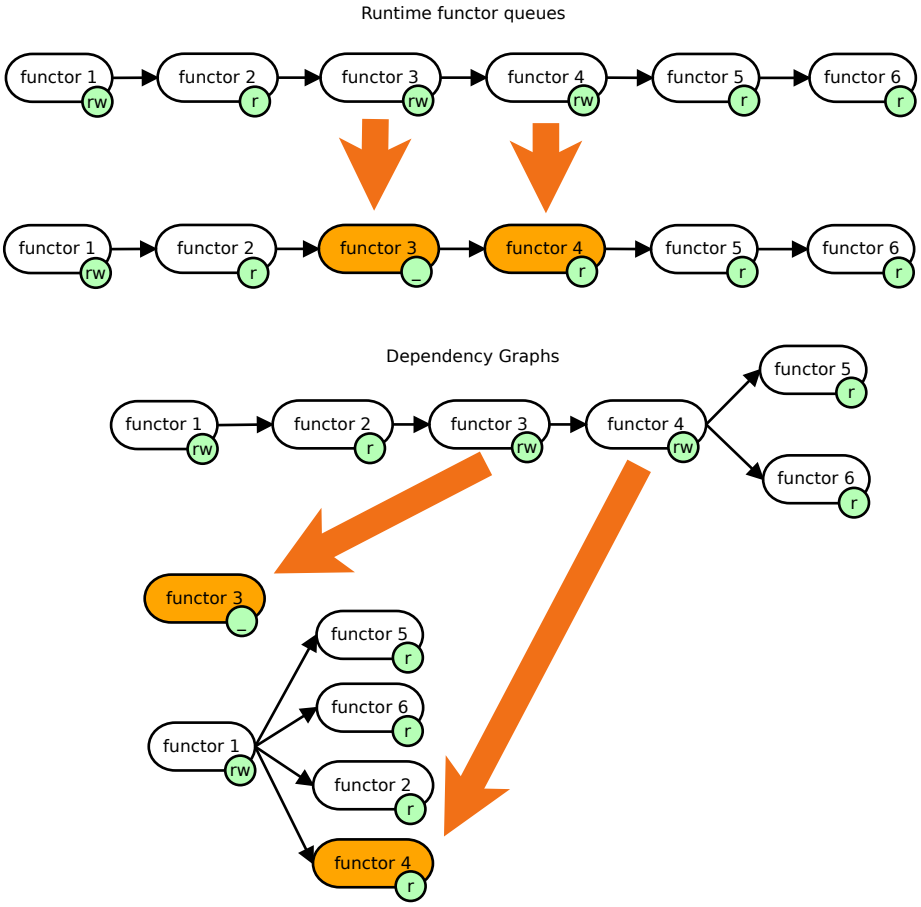


Figure 5: Demotion of two (sub) functors from read-writing to a resource to reading and doing nothing with it.

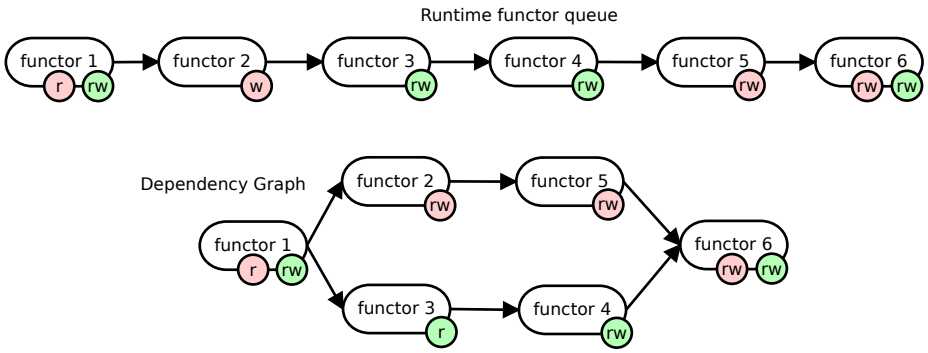


Figure 6: The dependency graph also support different kind of resources, here in red and green.

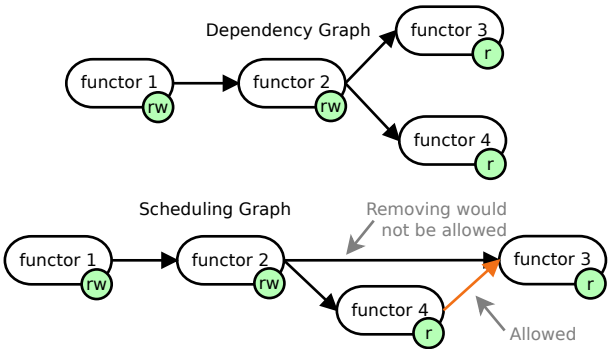


Figure 7: The scheduling graph has the very same nodes as the dependency graphs and all its edges. It is only allowed to add more edges. Although the edge between functor 2 and functor 3 is useless now, it is not allowed to remove it.

3.1 Scheduling flags

Besides dependencies built from the order of resource access, their may be other informations, which must be considered while scheduling the functors. An example are MPI calls, which need to be

- run on the main thread and
- need to keep the order if blocking MPI is used.

3.1.1 Binding a functor to the main thread

Especially the first constraint cannot be described directly with the resource access information, thus flags are introduced here. A flag is very similar to a resource access, but is constant for the whole lifetime of the functor (no demotion or promotion) and has a default value even if it is not explicit defined for a functor. For MPI such a flag could be e.g. “Schedule thread restriction” with the values “main thread” and “doesn’t matter”, which would result in a scheduling graph with further scheduling information as shown in figure 8 on the following page. The same mechanism could be extended to support hardware accelerators or heterogeneous multi cores, where functors have to run on a specific hardware.

3.1.2 Network access constraints

It is in discussion, how to abstract the second constraint about MPI. It could be abstracted as flag, but also as resource giving the possibility to differ between tags and MPI communicators. Both is possible and for the scheduler and the dependency graph it makes no difference.

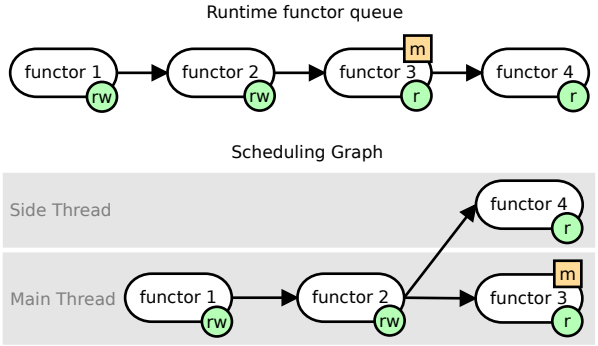


Figure 8: Use of flags to give some extra informations for the scheduler, in this case to tell the scheduler to have functor 3 scheduled on the main thread in any case.

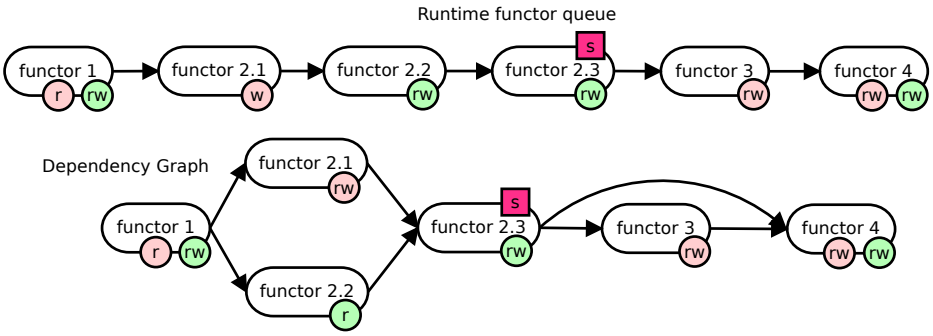


Figure 9: Synchronizing flag, which is directly interpreted by the graph creation algorithm and doesn't need to be implemented by the scheduler.

3.1.3 Synchronizing functors

Although it is always bad for scheduling to add a barrier, it may be needed from time to time. For this another kind of flag called "execution property" could be implemented with two states "independent" (default) and "synchronizing". The second status means, that every functor defined until now needs to be finished (including the synchronizing functor itself) before any other functor can start as seen in figure 9.

It may be a use case to have functor being synchronous to each other, but the other functors can run free concurrent, but it makes no sense to add this functionality to the flag as this can be implemented as mutual exclusive resource in the dependency graph.

3.2 Exclusive Resource without order

There exist algorithms, which write on the same resource, but do need to be scheduled necessarily in order. If e.g. a tiling board algorithm is started multiple times with different shifts to modify a whole array, the order doesn't matter. On the other hand do tiling board algorithm not access always the whole resource, so they may have different

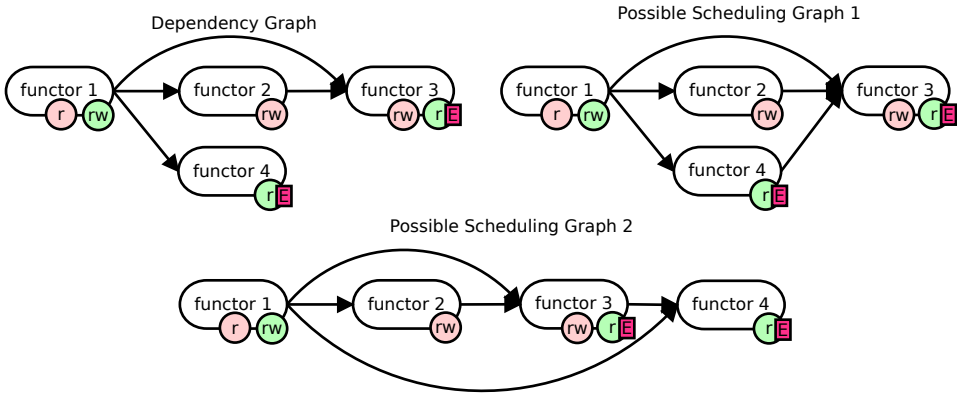


Figure 10: The Exclusive flag tells the scheduler that functor 3 and 4 most not run at the same time, but it is up to the scheduler to decide for an order of scheduling resulting in two different solutions. Solution 1 would probably make more sense, but solution 2 is valid nevertheless.

access areas (see section 4.2). A functor, which is independent from the center of the resource, but only works e.g. on the left border (like sending the border to the left neighbor) could be scheduled if a tiling board algorithm runs, which does not access this border.

Because of this it may make sense to also add (scheduling) flags to the resources itself, which could e.g. in this case state, that the scheduler must add edges between parallel resource access, but is free to decide the order based on other resource dependencies as seen in figure 10.

4 Functor Runtime states

4.1 Progress

While a functor is waiting and still have parents in the scheduling graph, it is called *pending*. If a functor does not have parents anymore, as they are finished, it gets the state *ready*. If the scheduler decides to run a ready functor, the states switches to *running*. When the functor/transition finished and all following places are activated, the functor is *done*.

Whenever a state is changed, it can easily be logged and written out by the scheduler or the dependency/scheduling graph manager. With this it is very easy and fast forward to create a detailed trace of a program run.

4.2 Resource List

Each functor has as list of resources that this functor will access in one way or an other, each resource has the states

	read	write	atomic add	atomic mul
read	**ind**	**dep**	**dep**	**dep**
write	**dep**	**dep**	**dep**	**dep**
atomic add	**dep**	**dep**	**ind**	**dep**
atomic mul	**dep**	**dep**	**dep**	**ind**

Table 1: Dependency of the row resource flags from the column resource flags (*ind* = independent, *dep* = dependent).

4.2.1 Resource ID

The *Resource ID* is an unique identifier identifying the resource. In the earlier images this was the color of the resource. This my be a local shared memory access, but resources can also be MPI communicators or similar.

4.2.2 Resource Access

The *Resource Access* describes, how the functor accesses this resource. This can be e.g *read_*write, *write* and *no_access*, but also an explicit separation of atomic operations make sense as commutative, atomic operations can work safely in parallel. The order of demotion could be e.g.:

$$\text{read\_write} \rightarrow \begin{matrix} \nearrow \\ \\ \searrow \end{matrix} \begin{matrix} \text{atomic\_add} \\ \text{atomic\_mul} \\ \text{read} \end{matrix} \begin{matrix} \searrow \\ \\ \nearrow \end{matrix} \rightarrow \text{no\_access}$$

. Atomic subtraction and division are just special varieties of addition and multiplication. This dependency can be abstracted as matrix. Table 1 shows the dependency of the resource flag types for the different access types.

But the resource access does not necessarily need to be implemented as simple flags, also range checks are possible e.g. if two functors work on the same resource, but not in the same area inside the resource. In that case they are independent.

5 Example Resource (in PIConGPU)

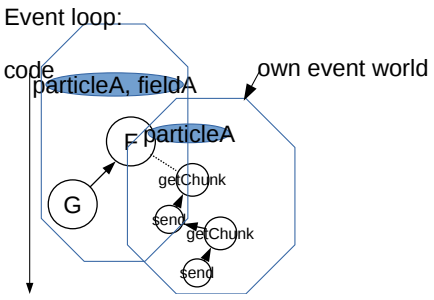
Some exemplary sources of PIConGPU could be:

- E-, B-, or J-field
- particle buffer electrons (border + core)
- particle buffer electrons (exchange buffer for send)
- particle buffer electrons (exchange buffer for receive)

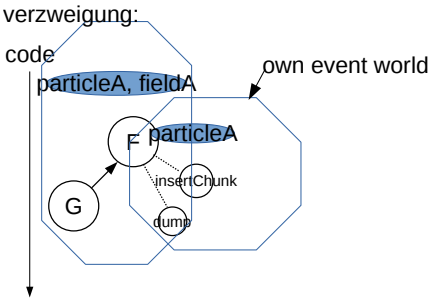
- double buffers either
  - as sub-resource or
  - as individual resources

6 Example Tasks in PMacc

```
1  //Event Loop
2  Ressource particle;
3  numParticles = 4242;
4  do {
5    //e.g. chunk with 100 particles
6    chunkSize.viewToParticleChunk =
7      getParticleChunkFromGPU( particle);
8    send( size, viewToParticleChunk );
9    numParticles -= chunkSize;
10 }
11 while( numParticles > 0 );
```




```
1  //RT branching
2  Ressource particle;
3  size.particleChunk = receiveChunk( );
4  if (size == 0)
5    do_nothing();
6  else if (size == maxChunkSize)
7    insertParticleChunk(
8      particleChunk ,particle );
9  else if (size < maxChunkSize)
10   dumpParticleChunk( particleChunk );
```



7 Possible implementation

This section shall give some implementation ideas and how the whole scheduling and graph building process can be abstracted in an open and future-proof way.

7.1 Access Concept

For this example an access concept is needed to export two methods is_dependent_of and is_superset_of, which both return a bool value. The first method takes another access of the same type and tells, whether the caller is dependent of the argument. The second one has the same argument and checks, whether the caller is a superset of the passed access. With this it can be checked, whether a demotion is valid or not.

7.1.1 Matrix Access Implementation

This code snippet shows, how it may be possible to abstract the mentioned resource access types based on matrices implementing the access concept. This solution is withal independent of a specific matrix, thus can be extended easily. The only requirement is, that Matrix::Entry is defined (which describes the row and column names) and that at least on Entry is called Matrix::Entry::None.

```
template <typename Matrix>
struct MatrixAccess
{
  using MatrixEntry = typename Matrix::Entry;
  inline MatrixAccess(MatrixEntry entry = MatrixEntry::None) :
    state(entry)
  {}
  inline bool is_dependent_of(MatrixAccess &right)
  {
    return Matrix::matrix[state][right.state];
  }
  inline bool is_superset_of(MatrixAccess &right)
  {
    //Checking, that no new dependencies are created
    for (unsigned i = 0; i < Matrix::size; i++)
      // false -> true is not allowed!
      if ( !Matrix::matrix[state][i] && Matrix::matrix[right.state][i])
        return false;
    return true;
  }
  MatrixEntry state;
};
```

A possible implementation for the template argument Matrix, which implements the already mentioned states “read-write”, “read”, two atomic writes and “none” (as it is required) can be seen here:

```
struct IOMatrix
{
    static constexpr std::size_t size = 5;
    enum Entry : unsigned { ReadWrite = 0, Read = 1, Atomic_Add = 2,
    Atomic_Mul = 3, None = 4 };
    static const bool matrix[size][size];
    static std::string to_string(Entry entry)
    {
    switch(entry)
    {
      case ReadWrite: return std::string("rw");
      case Read: return std::string("read");
      case Atomic_Add: return std::string("aadd");
      case Atomic_Mul: return std::string("amul");
      case None: return std::string("none");
    }
    return std::string("unknown");
  }
};
```

```
19
20 // row depends on column
21 const bool IOMatrix::matrix[IOMatrix::size][IOMatrix::size] = {
22   // row \ col rw     read    aadd    amul    none
23   /*  rw  */ {  true,  true,  true,  true, false },
24   /* read */ {  true, false,  true,  true, false },
25   /* aadd */ {  true,  true, false,  true, false },
26   /* amul */ {  true,  true,  true, false, false },
27   /* none */ { false, false, false, false, false }
28 };
```

7.1.2 Area Access Implementation

With this struct it is able to define a used area for a resource and to check for dependencies and the possibility to decrease the accessed area:

```
1  template <typename DimensionType, std::size_t dimensions>
2  struct AreaAccess
3  {
4    using PairType = std::pair<DimensionType,DimensionType>;
5    using PairTypeList = std::array<PairType,dimensions>;
6    inline AreaAccess(PairTypeList sizes) :
7      sizes(sizes)
8    {}
9    inline bool is_dependent_of(AreaAccess &right)
10   {
11     AreaAccess& left = *this;
12     for (unsigned i = 0; i < sizes.size(); i++)
13     {
14         //    [     ->|<-    ->|<-     ]
15       if (left.sizes[i].second < right.sizes[i].first ||
16         // ->|<-     ]         [     ->|<-
17         right.sizes[i].second < left.sizes[i].first)
18         return false;
19     }
20     return true;
21   }
22   inline bool is_superset_of(AreaAccess &right)
23   {
24     AreaAccess& left = *this;
25     //right shall be in left
26     for (unsigned i = 0; i < sizes.size(); i++)
27     {
28         // ->|<- ->|<-    ]      ]
29       if (left.sizes[i].first  > right.sizes[i].first ||
30         //    [      [    ->|<- ->|<-
31         left.sizes[i].second < right.sizes[i].second)
32         return false;
33     }
34     return true;
35   }
36   PairTypeList sizes;
37 };
```

7.1.3 Mixing of Accesses

A typical resource in memory – like a 3d array – will most probably make use of both already shown access types, which can easily be connected as seen here:

```
template <std::size_t dimensions>
struct FieldAccess
{
  using DimensionType = float;
  using PairTypeList = typename AreaAccess<DimensionType,dimensions>::
    PairTypeList;
  inline FieldAccess(PairTypeList sizes,IOMatrix::Entry entry) :
    ioResource(entry),
    areaResource(sizes)
  {}
  inline bool is_dependent_of(FieldAccess &right)
  {
    bool ioDep   =   ioResource.is_dependent_of(right.ioResource);
    bool areaDep = areaResource.is_dependent_of(right.areaResource);
    return ioDep && areaDep;
  }
  inline bool is_superset_of(FieldAccess &right)
  {
    bool ioSup   =   ioResource.is_superset_of(right.ioResource);
    bool areaSup = areaResource.is_superset_of(right.areaResource);
    return ioSup && areaSup;
  }
  MatrixAccess<IOMatrix> ioResource;
  AreaAccess<DimensionType,dimensions> areaResource;
};
```

7.2 Resource implementation

After defining the access types, the implementation of a resource using them is very easy and short:

```
template <unsigned dimensions>
struct FieldResource
{
  using ResourceType = FieldAccess<dimensions>;
  FieldResource(std::string name,typename ResourceType::PairTypeList sizes,
    IOMatrix::Entry entry) :
    acc(sizes,entry),
    name(name)
  {}
  ResourceType acc;
  std::string name;
};
```

7.3 Testing the implementation

The following code tests the implementation for some combinations of areas and access types:

```
int main()
{
  FieldResource<3> fieldA("fieldA",
    {{{0.0f,2.0f},
      {0.0f,2.0f},
      {0.0f,2.0f}}},
    IOMatrix::Entry::ReadWrite);
  FieldResource<3> fieldB("fieldB",
    {{{0.0f,2.0f},
      {1.0f,2.0f},
      {0.0f,2.0f}}},
  IOMatrix::Entry::Read);
  FieldResource<3> fieldC("fieldC",
    {{{0.0f,2.0f},
      {0.0f,0.9f},
      {0.0f,2.0f}}},
  IOMatrix::Entry::Atomic_Add);
  print_field(fieldA);
  print_field(fieldB);
  print_field(fieldC);
  print_dependency(fieldA,fieldA);
  print_dependency(fieldB,fieldB);
  print_dependency(fieldC,fieldC);
  print_dependency(fieldA,fieldB);
  print_dependency(fieldB,fieldA);
  print_dependency(fieldA,fieldC);
  print_dependency(fieldC,fieldA);
  print_dependency(fieldB,fieldC);
  print_dependency(fieldC,fieldB);
  print_superset(fieldA,fieldA);
  print_superset(fieldB,fieldB);
  print_superset(fieldC,fieldC);
  print_superset(fieldA,fieldB);
  print_superset(fieldB,fieldA);
  print_superset(fieldA,fieldC);
  print_superset(fieldC,fieldA);
  print_superset(fieldB,fieldC);
  print_superset(fieldC,fieldB);
}
```

with these helper functions:

```
template <typename Field>
void print_field(Field const & field)
{
  std::cout << field.name << ": " <<
  "[" << std::to_string(field.acc.areaResource.sizes[0].first ) << ", " <<
         std::to_string(field.acc.areaResource.sizes[0].second) << "] " <<
  "[" << std::to_string(field.acc.areaResource.sizes[1].first ) << ", " <<
         std::to_string(field.acc.areaResource.sizes[1].second) << "] " <<
  "[" << std::to_string(field.acc.areaResource.sizes[2].first ) << ", " <<
         std::to_string(field.acc.areaResource.sizes[2].second) << "] " <<
  IOMatrix::to_string(field.acc.ioResource.state) << std::endl;
}

```

```
14 template <typename Field>
15 void print_dependency(Field & left,Field & right)
16 {
17   std::cout << "Dependency " << left.name << " -> " << right.name <<
18   " (== they need to run sequentially): " <<
19   std::to_string(right.acc.is_dependent_of(left.acc)) << std::endl;
20 }
21
22 template <typename Field>
23 void print_superset(Field & left,Field & right)
24 {
25   std::cout << left.name << " is superset of " << right.name <<
26   " (== " << left.name << " can be demoted to " << right.name << "): " <<
27   std::to_string(left.acc.is_superset_of(right.acc)) << std::endl;
28 }
```

The output looks likes this:

```
1  fieldA: [0.000000, 2.000000] [0.000000, 2.000000] [0.000000, 2.000000] rw
2  fieldB: [0.000000, 2.000000] [1.000000, 2.000000] [0.000000, 2.000000] read
3  fieldC: [0.000000, 2.000000] [0.000000, 0.900000] [0.000000, 2.000000] aadd
4  Dependency fieldA -> fieldA (== they need to run sequentially): 1
5  Dependency fieldB -> fieldB (== they need to run sequentially): 0
6  Dependency fieldC -> fieldC (== they need to run sequentially): 0
7  Dependency fieldA -> fieldB (== they need to run sequentially): 1
8  Dependency fieldB -> fieldA (== they need to run sequentially): 1
9  Dependency fieldA -> fieldC (== they need to run sequentially): 1
10 Dependency fieldC -> fieldA (== they need to run sequentially): 1
11 Dependency fieldB -> fieldC (== they need to run sequentially): 0
12 Dependency fieldC -> fieldB (== they need to run sequentially): 0
13 fieldA is superset of fieldA (== fieldA can be demoted to fieldA): 1
14 fieldB is superset of fieldB (== fieldB can be demoted to fieldB): 1
15 fieldC is superset of fieldC (== fieldC can be demoted to fieldC): 1
16 fieldA is superset of fieldB (== fieldA can be demoted to fieldB): 1
17 fieldB is superset of fieldA (== fieldB can be demoted to fieldA): 0
18 fieldA is superset of fieldC (== fieldA can be demoted to fieldC): 1
19 fieldC is superset of fieldA (== fieldC can be demoted to fieldA): 0
20 fieldB is superset of fieldC (== fieldB can be demoted to fieldC): 0
21 fieldC is superset of fieldB (== fieldC can be demoted to fieldB): 0
```

8 Further Literature

Scheduling/Concurrency Patterns:
https://www.dre.vanderbilt.edu/~schmidt/POSA/POSA2/conc-patterns.html