Refactor phys_buffer

Refactor phys_buffer.F90

Current phys_buffer API:

- pbuf_defaultopts. ?pbuf_setopts : A single namelist variable pbuf_global_allocate is provided which makes the scope of all pbuf variables persistant
- pbuf_init : nullifies the pbuf structure
- pbuf_add : add a field to pbuf - scope can be physpkg or global, fields are added by name, an index into the buffer is returned
- pbuf_print: print a summary of fields in the pbuf to the output logfile
- pbuf_get_fld_name: Given an index return the field name
- pbuf_get_fld_idx: Given a field name return the field idx
- pbuf_old_tim_idx: Return the index of the oldest time level
- pbuf_next_tim_idx: Return the idx of the next time level
- pbuf_update_tim_idx : cycles the time index
- pbuf_allocate: Allocates space for the pbuf, all calls to pbuf_add must have occured prior to this call.
- pbuf_setval: Set the value of a pbuf field to an r8 constant value
- pbuf_init_restart, pbuf_write_restart, pbuf_read_restart : Interface to cam restart files.

phys_buffer also provides public access to the derived pbuf field so there is code throughout cam physics that looks like:

```
real(r8), pointer, dimension(:,:) :: iciwpconv

iciwpconv  => pbuf(iciwpconv_idx)%fld_ptr(1,1:pcols,1:pver,lchnk, 1)
```

Limitations of current phys_buffer API:

The current pbuf API has several limitations that we would like to overcome.

1. Only r8 variables are allowed, this should be extended to allow integers
2. Dimensionality of variables is limited - the cam physics decomposition has pcols in the innermost dimension and chunks in the outermost. Current design allows for a dimension inside pcols (almost never used), a dimension outside chunks (again almost never used) and a single dimension between them which is frequently overloaded to support 2 or more actual dimensions needed in this location. The new API should remove the options of inner and outer dimensions and allow for a variable number of middle dimensions from 0 to some fixed but easily modifiable maximum.
3. The pbuf structure itself should not be public, functions in this module should provide pointers to or copies of individual fields in the structure.

> *(eaton, 4/5/11) I agree with this.*

Comparison to other similar functionality:

The functionality of cam_history_buffers is very similar to that of phys_buffers - I would propose that a replacement be created with an eye toward replacing both.

> *(eaton, 4/5/11) Not sure about this. Do you think that cam_history could directly use the pbuf?*

(edwards, 4/8) I did, but the current discussion has moved away from that idea.

Proposed new API:

- ~~pbuf_add_dim(dimname, dimval, dimid) -- Predefine dims for pcols, plev, plevp and chunks~~
- ~~pbuf_add_var(varname, persistance, var_type, decomp, (/dimid1, dimid2, dimid3/), index)~~
- pbuf_add_var(varname, persistance, var_type, (/dimlen2,dimlen3, .../)) Declare a new variable in pbuf - the first dimension (pcols) and the last (chunks) are implied.
- pbuf_get_var_ptr (name or index, lchunk) This function would return a pointer to a pbuf field of defined dimensionality (overloaded for # of dims and type)
  - pbuf_get_var_ptr_byname
  - pbuf_get_var_ptr_byid
- ~~pbuf_get_var_copy -- This subroutine would return a copy of the requested field (overloaded for # of dims and type)~~

> *(eaton, 4/5/11)*

I don't see a need for pbuf_add_dim. That's a cam_history kind of functionality that isn't needed for the pbuf. The user of pbuf should never need to refer to the pbuf dimensions by name, or by an id. Only by size, just like for a Fortran array.

What is the 'decomp' arg of pbuf_add_var for?

I think it's reasonable to assume that pcols will be the first index of the data array. But I'm wondering whether we want to include the begchunk:endchunk dimension as part of the data array, or whether we should allocate the pbuf array with a begchunk:endchunk dimension so that it could be treated more like the physics state.

- *if we use pbuf(:)%fld_ptr(...,begchunk:endchunk) then the user needs to supply the chunk index to access the data.*
- *if we use pbuf(:,begchunk:endchunk)%fld_ptr(...) then we can pass the chunk array sections from a high level (like state), and the user wouldn't need to be constantly accessing the chunk index from state in order to access info in the pbuf. If we do this then pbuf would no longer be a module variable, rather it would be something defined in a high level routine - cam_comp.F90 as is phys_state. But I don't think it should be pbuf(:, begchunk:endchunk) - I think I'd rather see pbuf (begchunk:endchunk)%field(:)%fld_ptr*

I'm not sold on pbuf_get_var_copy. I haven't seen any need for this to date. I'd prefer to have a minimal interface and require the user to copy data outside the pbuf interface if that's needed.

I think the pbuf_get method should return a pointer to the entire field by default, but provide optional array args, start/end, to allow access to an array subsection.

(eaton, 4/6/11)
Our discussion today largely focussed on the restart capability. One of the things I really like about the original pbuf design is that the user didn't need to be concerned about restart – as long as the pbuf field was 'global' scope it would end up in the restart file. I'd like to maintain this simplicity which is why I'm not in favor of the pbuf_add_dim method which requires the user to think about something that's only relevant for the restart file.

Here's another idea. What about removing all the restart methods from the pbuf module and just passing pbuf as an argument to the restart_physics methods. Let restart_physics be entirely responsible for getting the global pbuf fields to the restart file. This would be similar to what's currently done for the cam_out export fields which are also dealt with directly by the restart_physics methods.

Franis has convinced me that 'physpkg' and 'global' scope are not as obvious as using a terminology that implies fields are persistent or not (like the argument in Jim's proposed pbuf_add_var method). Scope is more a programming concept that most physical scientists aren't familiar with.