The user calls, on thread 0, the function CkLoop\_ParallelizeHybrid() providing the function to be called, the range of iterations of the loop and the number of chunks of the loop. This function calls the function parallelizeFuncHybrid() of the global FuncCkLoop object.

**parallelizeFuncHybrid():**

The object curLoop has information about the current loop that is shared across all threads. This function sets the information in curLoop. It creates a converse message with curLoop as a field and hybridHandler as the index of the function to call where it is executed.

The message is then enqueued in the regular queue of all the PEs except 0. At this point, cfunc is called if it is set. The function then calculates the variable lastStatic and the variable lastDynamic for itself, i.e., PE 0.

The function enqueues each dynamic chunk separately in the task queue of itself using CsdTaskEnqueue. The field numDynamicChunksFired is incremented by j, the number of dynamic chunks enqueued (it must be an atomic variable).

The function then invokes the user-provided function *func* on PE 0’s static range.

To satisfy termination condition, the function increments curLoop->numStaticRegionsCompleted. Then, the function dequeues PE 0’s own dynamic chunks until PE 0’s task queue is emptied and executes the user function for each chunk via CmiHandleMessage(msg). The function then invokes the function waitLoopDoneHybrid() to wait until the other cores finish their static and dynamic chunks. Other cores steal and help each other with dynamic chunks, but core 0 does not help in the current version of the CkLoopHybrid library.

If the user specifies a reduction, the values produced by all chunks/threads must be reduced at this point.

What is happening on the other cores?

The func hybridHandlerFunc is called on each core. That function calls the function doWorkForMyPe().

**doWorkForMyPe():**

The function calculates the beginning index of the static section, the beginning index of the dynamic section, etc.

The function then enqueues each dynamic chunk separately in the task queue of itself using CsdTaskEnqueue. The field numDynamicChunksFired is incremented by j, the number of dynamic chunks enqueued (it must be an atomic variable).

We then invoke the user-provided function *func* on PE 0’s static range.

To satisfy termination condition, we increment curLoop->numStaticRegionsCompleted. It also increments finishFlag,which isn’t necessary.

After doing this, control goes back to the converse scheduler, which will execute the dynamic chunks and steal from other cores as necessary.

**waitLoopDoneHybrid()** called from core 0:

while ((numStaticRegionsCompleted != numHelpers) || (numDynamicChunksCompleted != numDynamicChunksFired)) { do nothing};

This works if/because the variables involved are atomic.

if (type!=CKLOOP\_NONE)

    reduce(curLoop->getRedBufs(), redResult, type, numChunks);

return;

#define COMPUTE\_REDUCTION(T) {\

for(int i=0; i<numChunks; i++) {\

result += \*((T \*)(redBufs[i])); \

/\*CkPrintf("CkLoop Reduce: %d\n", result);\*/ \

}\

}

**singleHelperStealWork(msg)**

loopRec \* msg->loopRecPtr;

Enqueue dynamic chunks in your own taskQueue.

Do dynamic first. Once you start static, others should be ready to steal from your dynamic.

Work on your static chunks by calling the helperFunc which is written by the user.

**stealWork():**