Multiparadigm Parallel Programming with Charm++, Featuring ParFUM as a case study

5th Annual Workshop on Charm++ and its Applications

Aaron Becker abecker3@uiuc.edu UIUC 18 April 2007







There are lots of ways to write a parallel program

BSP X10

Global Arrays

OpenMP

High Performance Fortran

Fortress

Parallel Matlab

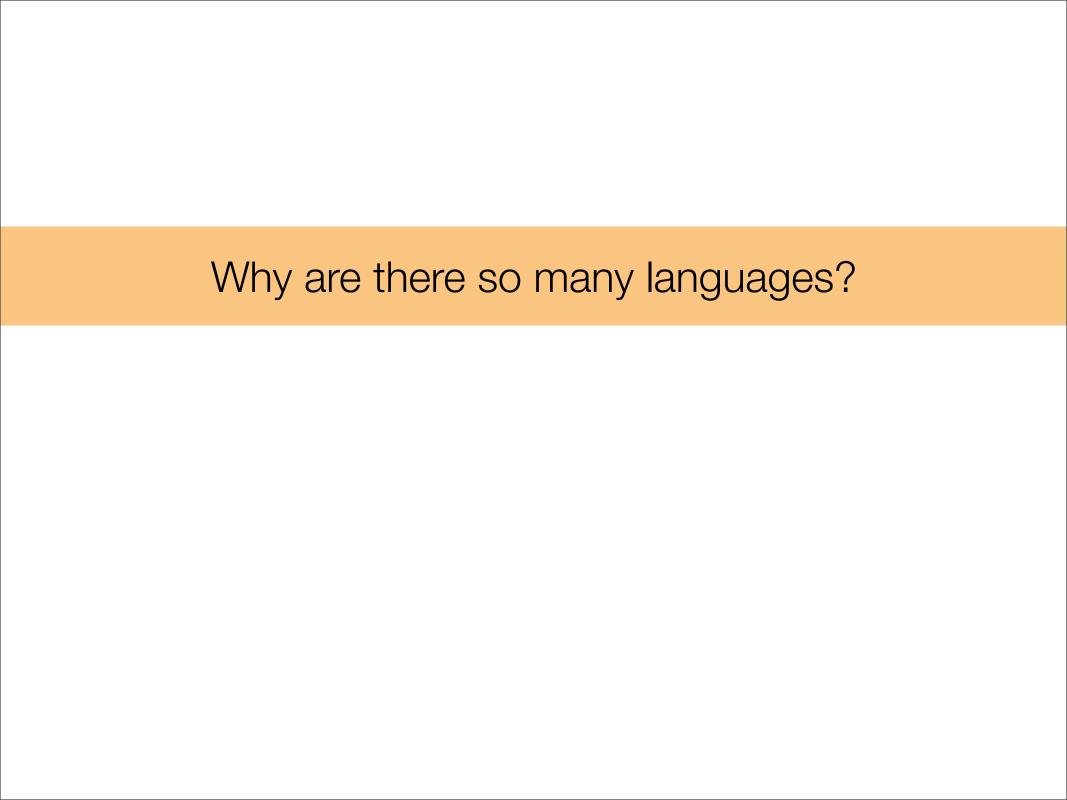
MPI

Charm++

Multiphase Shared Arrays

Chapel

Unified Parallel C

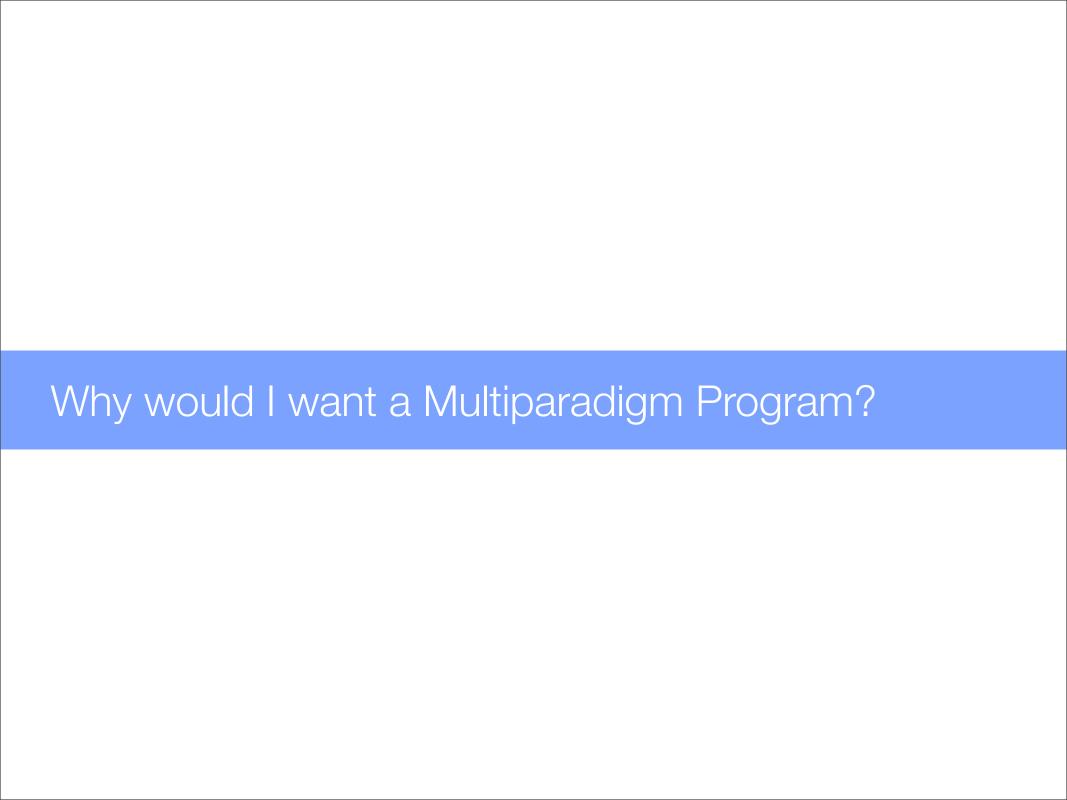


Why are there so many languages?

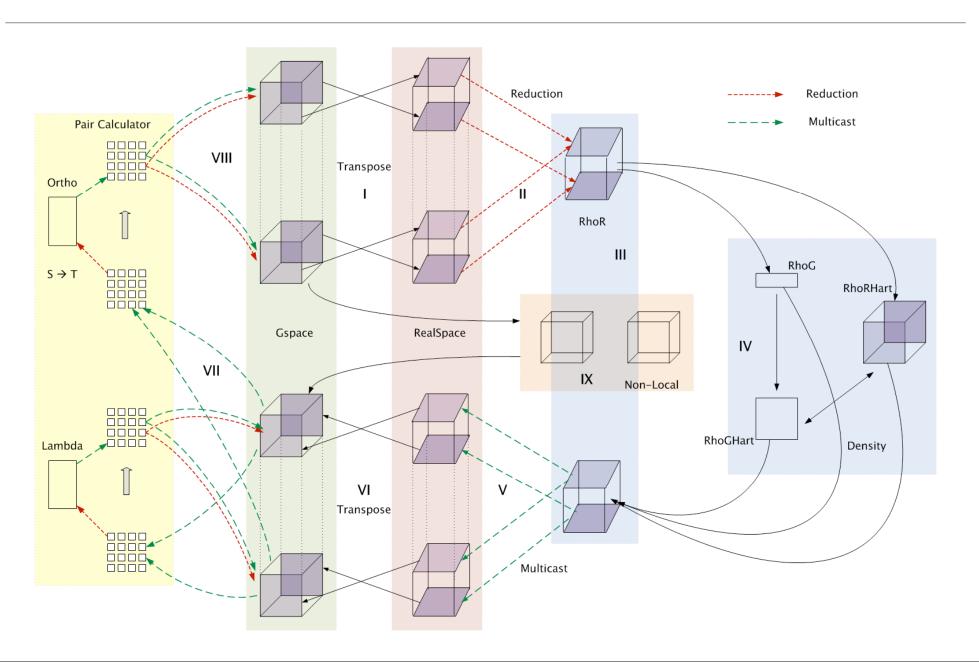
Each is good at something different

Automatic parallelizing of loops Fine-grained parallelism Unpredictable communication patterns So, what is a multiparadigm program?

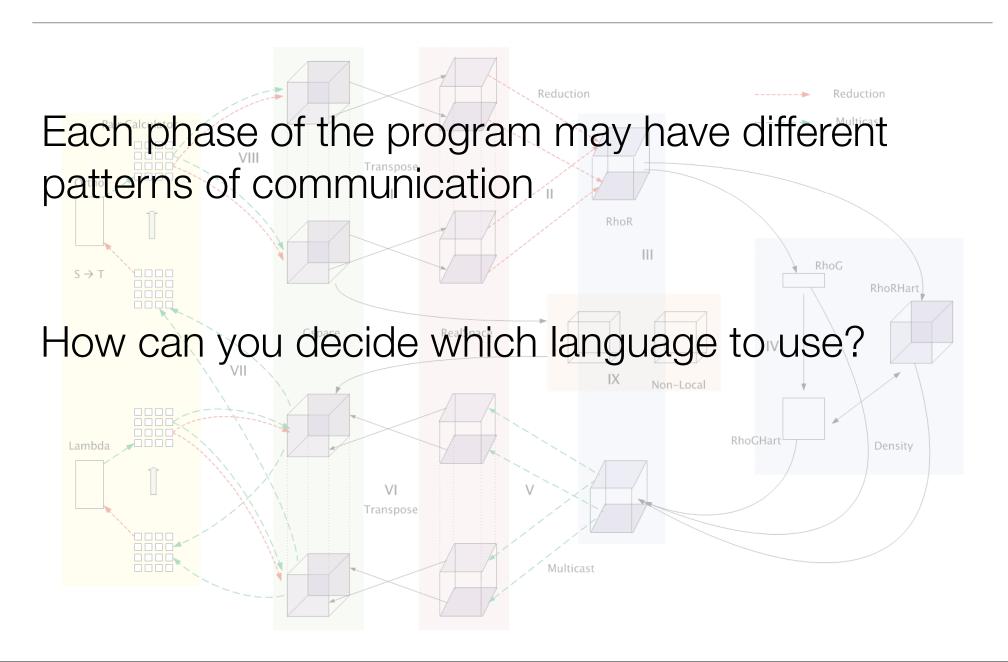
A program composed of modules, where each module could be written in a different language



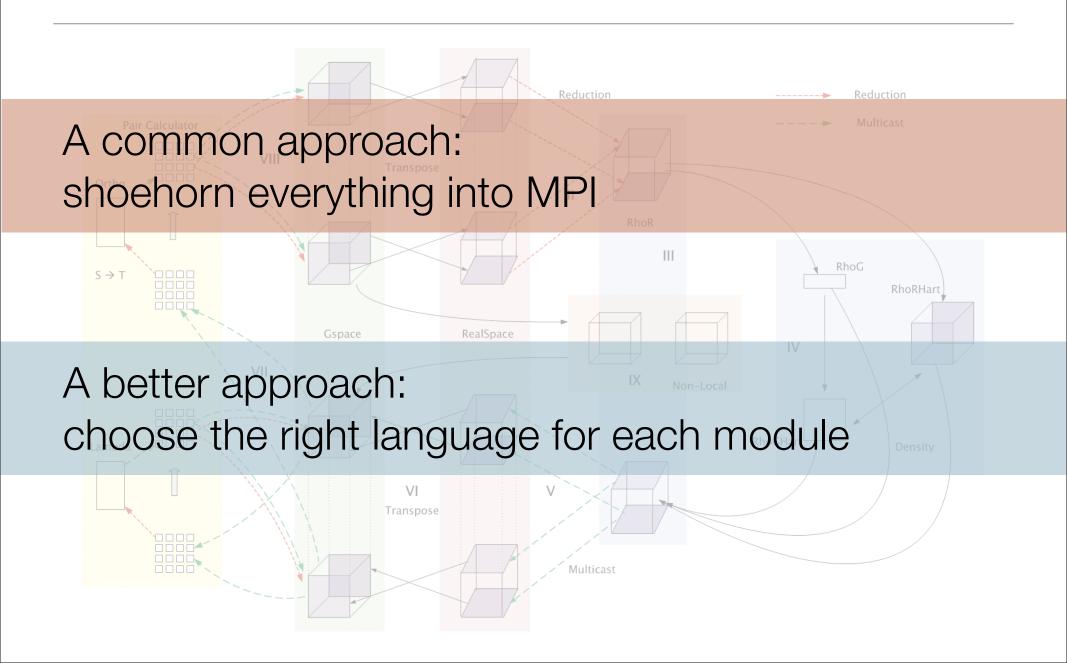
Suppose you have a complex program to parallelize



Suppose you have a complex program to parallelize



Suppose you have a complex program to parallelize



Suppose you have an existing MPI program

You want to add a new module, but it will be tough to write in MPI

Suppose you have an existing MPI program

You want to add a new module, but it will be tough to write in MPI

A common approach: write it in MPI anyway

A better approach: choose a better suited language

Why aren't multiparadigm programs more common?

Multiparadigm programs are hard to write: You need a way to stick these modules together

It's relatively simple if you only have one language in use at once: MPI/OpenMP hybrid codes run just one language at a time

For tightly integrated codes with multiple concurrent modules, you need a runtime system to manage them all

Where does Charm++ fit in?

The Charm++ runtime system (RTS) handles most of the difficulties of multiparadigm programming

Modules using different languages are co-scheduled and can integrate tightly with one another

The RTS supports several languages We are interested in adding more



What is ParFUM?

ParFUM: a Parallel Framework for Unstructured Meshing

Meant to simplify the development of parallel unstructured mesh codes

Handles partitioning, synchronization, adaptivity, and other difficult parallel tasks

ParFUM is multiparadigm

ParFUM consists of many modules, written in a variety of languages. I will briefly present three examples:

Charm++ for asynchronous adaptivity

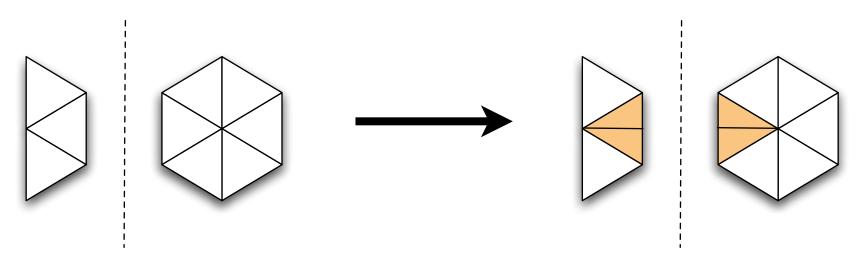
Adaptive MPI for the user's driver code and glue code to connect modules

Multiphase shared arrays (MSA) for data distribution

Charm++ in ParFUM

Asynchronous Incremental Adaptivity

Local refinement or coarsening of the mesh, without any global barriers.



Edge bisection on a processor boundary

What is Charm++?

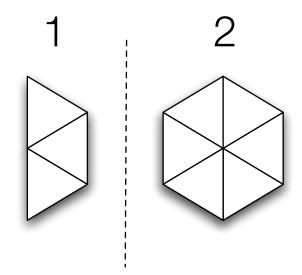
I hope you attended the fine tutorial by Pritish Jetley and Lukasz Wesolowski

In a nutshell, parallel objects which communicate via asynchronous method invocations

Why is Charm++ good for incremental adaptivity?

Incremental adaptivity leads to unpredictable communication patterns.

Suppose a boundary element of partition 1 requests refinement



How will partition 2 know to expect communication from 1? In MPI, this is very hard. In Charm++, it is natural.



What is Adaptive MPI?

For our purposes, it's just an implementation of MPI on top of the Charm++ RTS

For more information, see Celso Medes's tutorial on Friday at 3:10, How to Write Applications using Adaptive MPI

Why is Adaptive MPI important in ParFUM?

User provided driver code

Glue code between modules

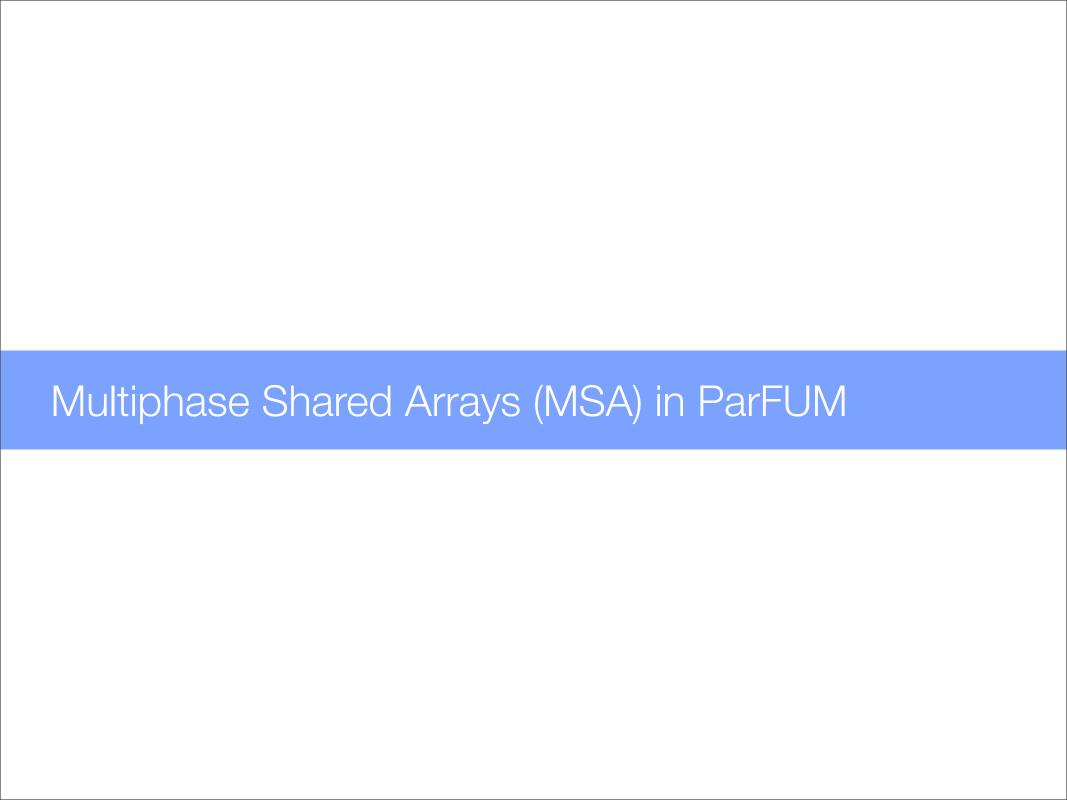
Why is Adaptive MPI important in ParFUM?

User provided driver code

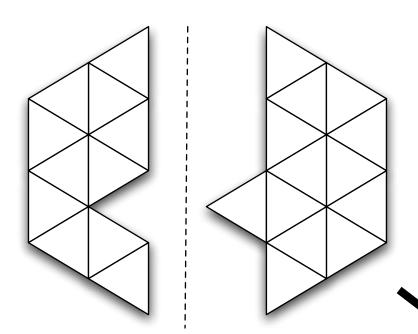
Popularity Legacy

Glue code between modules

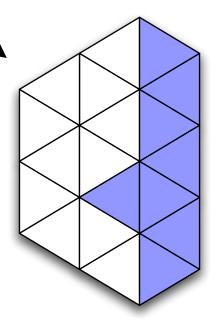
Simple flow of control

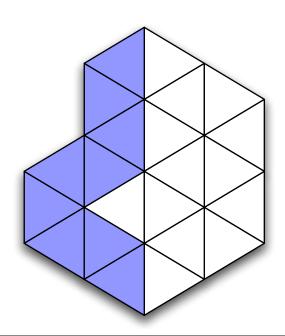


A data distribution problem

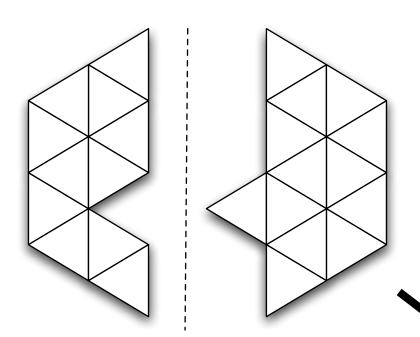


After initial partitioning, we need to determine which boundary elements must be exchanged.



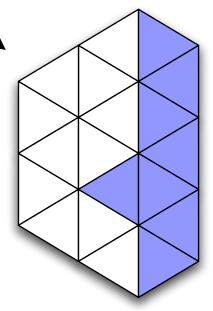


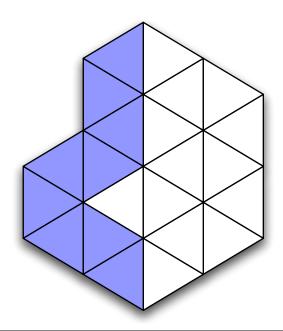
A data distribution problem



After initial partitioning, we need to determine which boundary elements must be exchanged.

What we would like: an easily accessible global table to look up shared edges





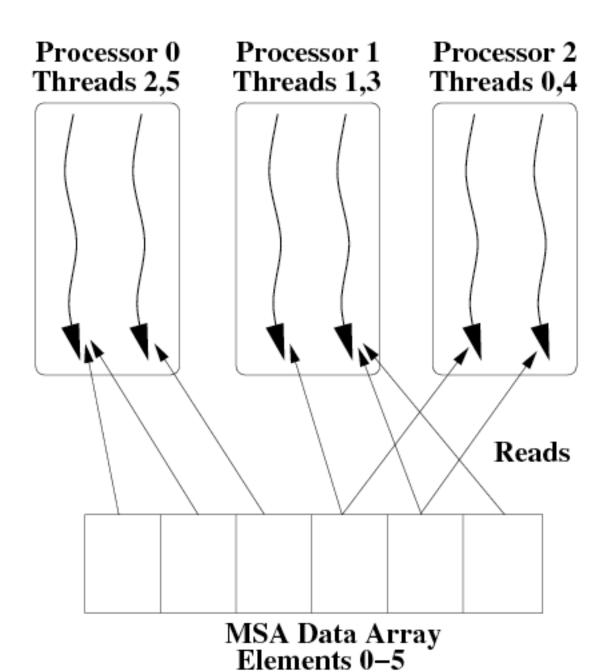
What is MSA?

Idea: shared arrays, where only one type of access is allowed at a time

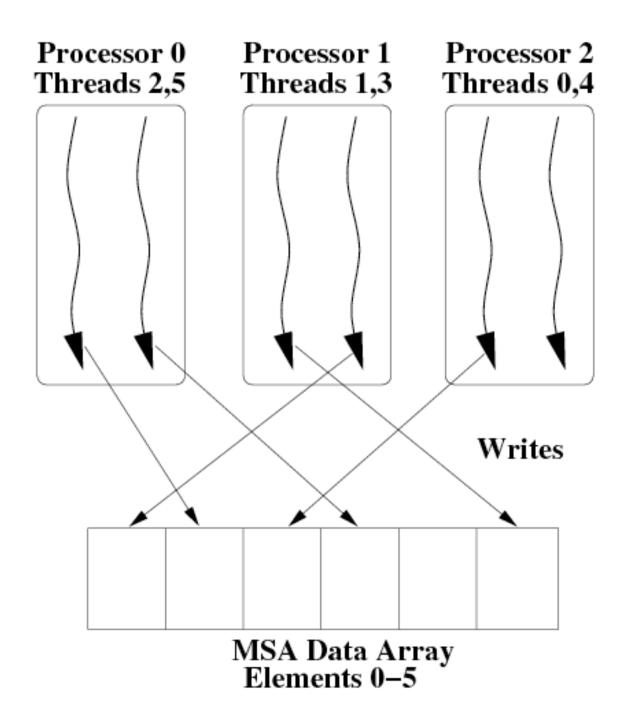
Access type is controlled by the array's phase

Phases include:

read-only write-by-one accumulate

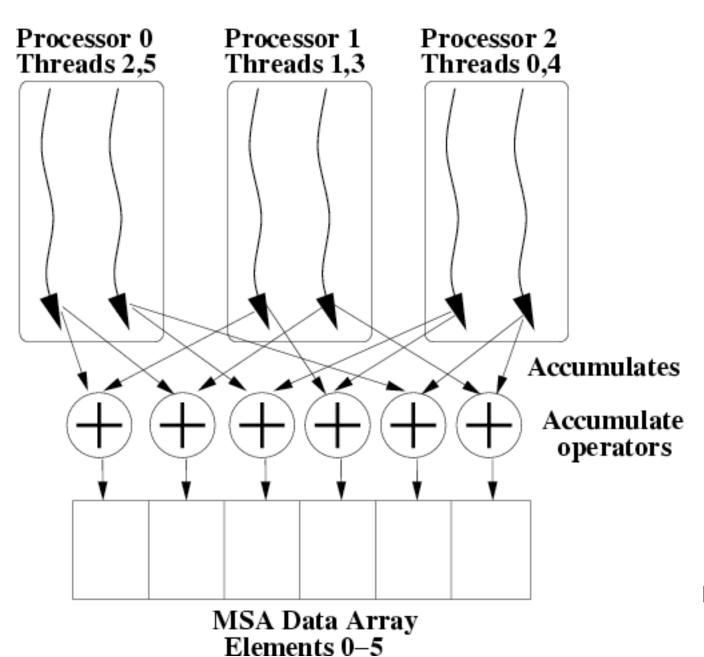


Read-only mode



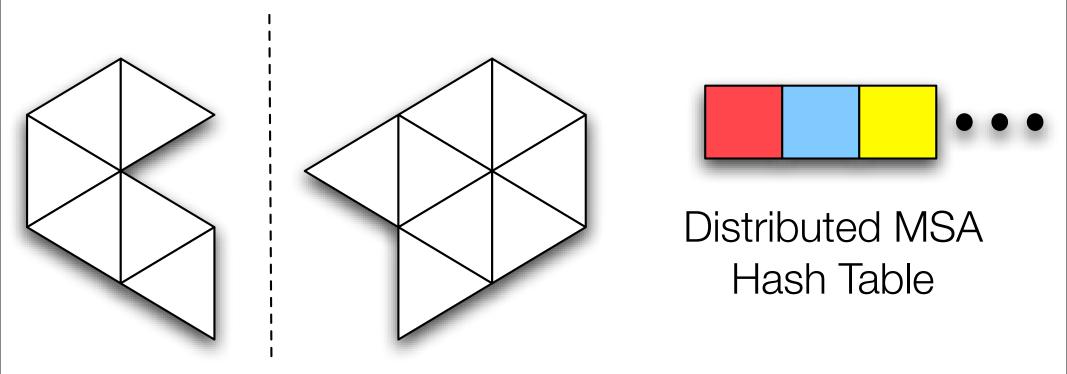
Write-by-one mode

note: one thread could write to many elements

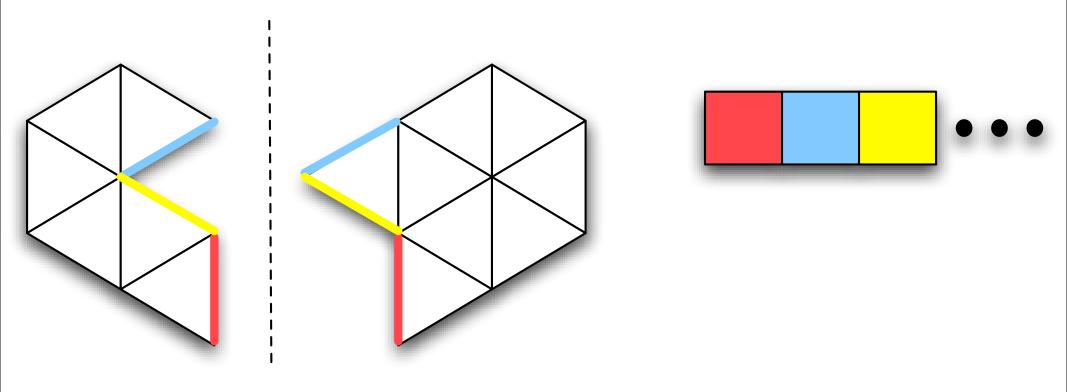


Accumulate mode

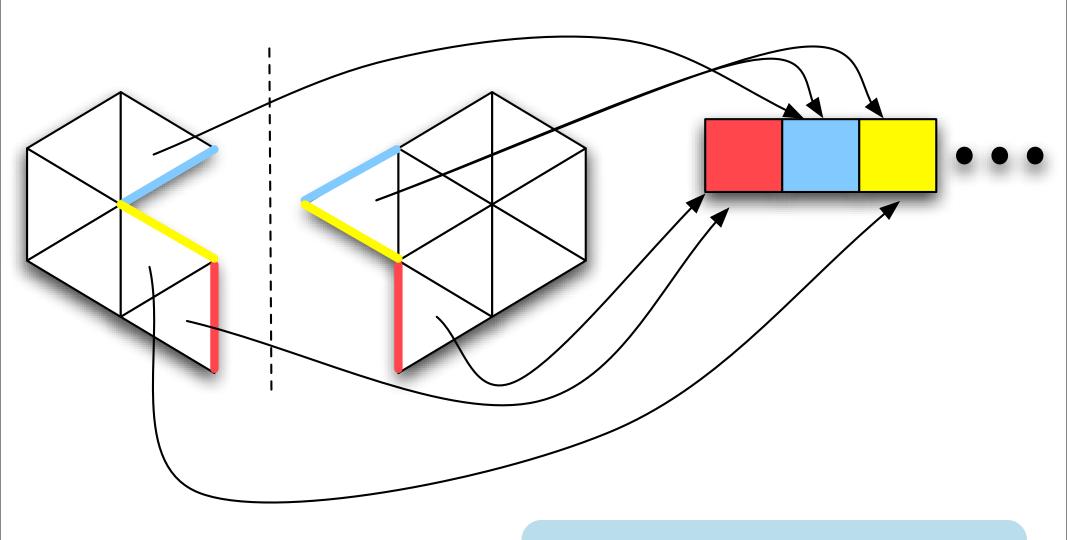
note: accumulation operator must be associative and commutative



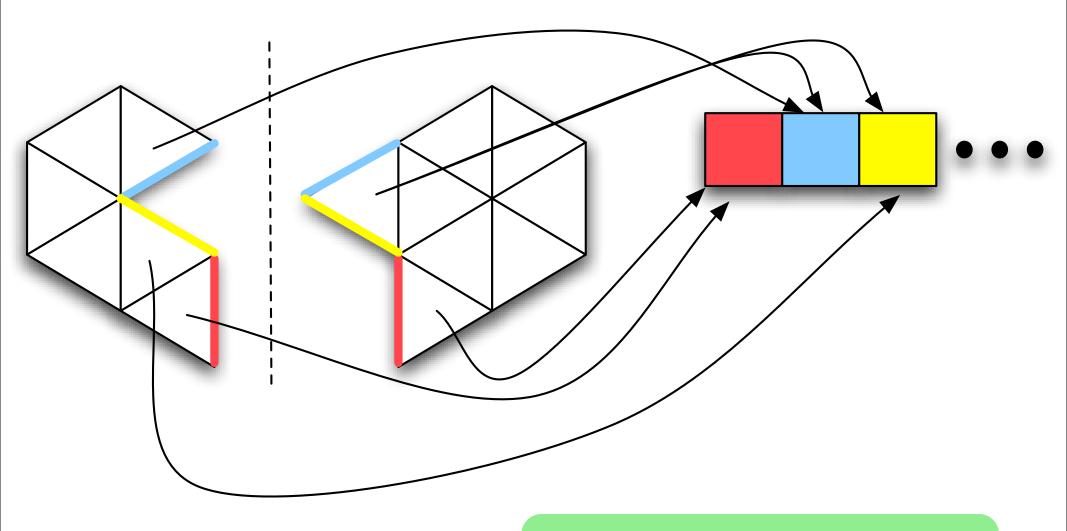
Partitioned Mesh



Each shared edge is hashed



Entries are added to the table in accumulate mode



Now elements which collide in the table probably share an edge

Why is MSA good for this application?

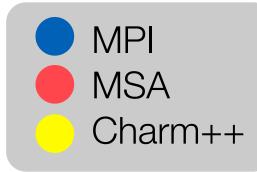
Shared access to a global table is convenient when trying to determine which partitions you need to send to or receive from

Filling and consulting the array fit neatly into MSA phases

How does this look in practice?

```
MPI_Comm_rank((MPI_Comm)comm_context,&myRank);
//printf("[kd] FEM_Mesh_Parallel_broadcast called for mesh %d\n",myRank,fem_mesh);
int new_mesh
     if(mvRank == masterRank){
        '/' am the master, i have the element connectivity data and need
'/to send it to everybody
printf("|sd| Memory usage on vp 0 at the begining of partition %d \n",CkMyPe(),CmiMemoryUsage());
        new_mesh=FEM_master_parallel_part(fem_mesh,masterRank,comm_context);
        new_mesh=FEM_slave_parallel_part(fem_mesh,masterRank,comm_context);
      //temp to keep stuff from falling apart
                        clearPartition():
    //printf("[%d] Partitioned mesh number %d \n",myRank,new_mesh);
return new_mesh;
 int FEM_master_parallel_part(int fem_mesh,int masterRank,FEM_Comm_t comm_context){
   const char *caller="FEM_Create_connmsa";
     FEMAPI(caller):
     FEM_chunk *c=FEM_chunk::get(caller);
FEM_Mesh *m=c->lookup(fem_mesh,caller);
m->setAbsoluteGlobalno();
     int nelem = m->nElems():
     int numChunks:
    MFI_Comm_size((MFI_Comm)comm_context,&numChunks);
printf("master -> number of elements %d \n",nelem);
     DEBUG(m->print(0)):
     /*load the connectivity information into the eptr and
        eind datastructure. It will be read by the other slave elements and used to call parmetis*/
     MSAIDINT eptrMSA(nelem,numChunks);
MSAIDINT eindMSA(nelem*10,numChunks);
        after the msa array has been created and loaded with connectivity data
     struct conndata data;
     data.nelem = nelem;
data.nnode = m->node.size();
data.arr1 = eptrMSA;
data.arr2 = eindMSA;
     MPI_Bcast_pup(data,masterRank,(MPI_Comm)comm_context);
     eindMSA.enroll(numChunks);
     int indcount=0,ptrcount=0;
for(int t=0;t<m->elem.size();t++){
  if(m->elem.has(t)){
           FEM Elem &k=m->elem(t):
           for(int e=0;e<k.size();e++){
                                                  eptrMSA.set(ptrcount)=indcount;
                                                  for(int n=0;n<k.getNodesPer();n++){
 optrMSA.set(ptrount) = indcount;
printf("master -> ptrount %d indcount %d sizeof(MSAIDINT) %d sizeof(MSAIDINTLIST) %d memory %d\n",ptrount,indcount,sizeof(MSAIDINT),sizeof(MSAIDINT)
call parmetis
printf("done with parmetis %d FEM_Mesh %d in %.61f \n",CmiNemoryUsage(),sizeof(FEM_Mesh),CkWallTimer()-parStartTime);
      double dataArrangeStartTime = CkWallTimer();
int totalNodes = m->node.size();
MSAIDNWILIST nodepart(totalNodes,numChunks);
MSAIDNWILIST nodepart(totalNodes,numChunks);
MDI_Beast_pup(nodepart,nasterRank,(MPI_Comm)comm_context);
nodepart.enroll(numChunks);
FEM write nodepart(nodepart.partdata,(NPI_Comm)comm_context);

nvin+fi'Creating mapping of mode to partition took %.61f\n',CKWallTimer()-dataArrangeStartTime);
WRAIDNOBELIST part2node(numChunks,numChunks);
MPI_Bcast_pup(part2node,masterRank,(MPI_Comm)comm_context);
part2node.enroll(numChunks);
FEM write part2node(nodepart,part2node,partdata,(MPI Comm)comm context);
Partitioning
```



```
enum {FEM_globalID=33};
void femMeshModify::ckJustMigrated(void) {
   ArrayElement1D::ckJustMigrated();
              Arraylement|D:ck/ustKigrated();
//set the pointer to fabble
//set the pointer to fabble
Commission of the commission of 
                      setPointersAfterMigrate(fmMesh):
                 oid femNeshModify::setPointersAfterMigrate(FEM_Mesh *m) {
    faMesh * m;
    falign->FEM_InterpolateSetMesh(faMesh);
    fahdapt->FEM_AdaptSetMesh(faMesh);
    fahdapt->FEM_AdaptSetMesh(faMesh);
                      fmAdaptAlgs=>FEM_AdaptAlgsSetMesh(fmMesh);
for(int i=0; i<fmLockN.size(); i++) fmLockN[i].setMeshModify(this);</pre>
                 oid femMeshModify::setFemMesh(FEMMeshMsq *fm) {
           famenh = fm->m;

to = fm-bi;

to = fm-bi;

famenh-westFenney())

famenh-westFenneshenbodity(this);

famenh-westFenneshenbodity(this);

famenh-westFenneshenbodity(this);

famenh-westFenneshenbodity(fmenh, this);

famenh-westFenneshenbodity(fmenh, this);

int maine = fmenh-woode, size();

finneshenbodity(fmenh, this);

famenh-westFenneshenbodity(fmenh, this);

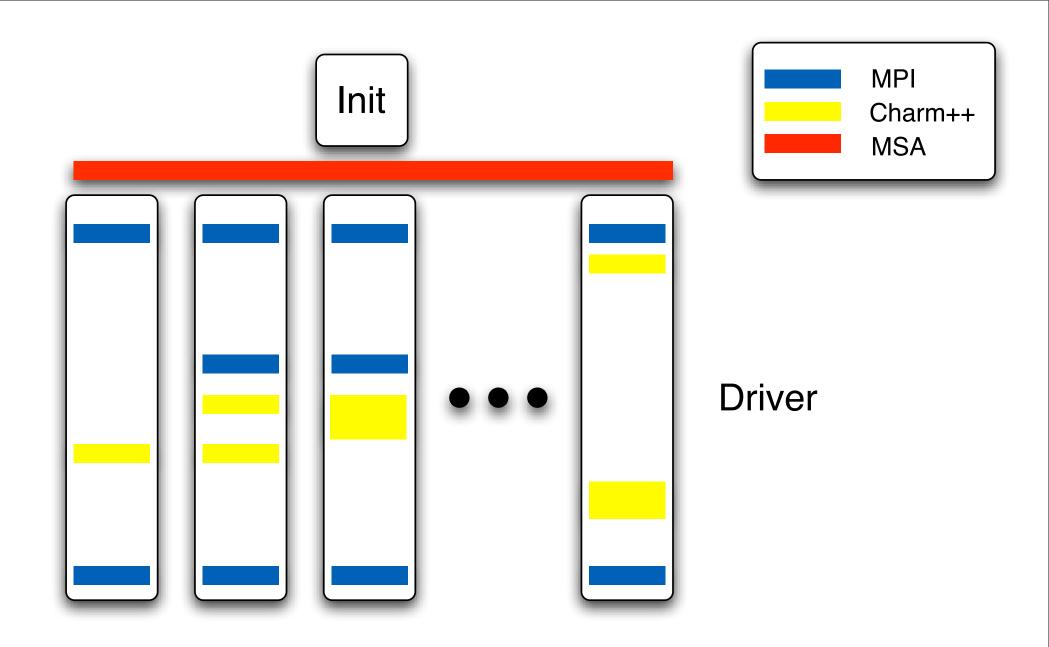
famenh-westFenneshenbodity(fmenh, this);

famenh-westFenneshenbodity(fmenh, this);
                                  for(int i=0; icqsize; i++) {
fmgLockN.push_back(new FEM_lockN(FEM_To_ghost_index(i),this));
                 }
//compute all the fixed nodes
for(int i=0; i<nsize; i++) {
   if(fmAdaptL->isCorner(i)) {
     fmfixedNodes.push_back(i);
}
/* Coarse chunk locks "/
inithag 'femmieshhodify:lock/memotechunk(int2Msg *msg) {
    "inithag 'femmieshhodify:lock/msg-bi;
    inithag 'insg new inithsg(0);
    init ret = finiock-block(msg-bi, msg-b-j);
    imsg-bi = grid
    delete msg;
    return imsg

  intMsg *femMeshModifv::unlockRemoteChunk(int2Msg *msg) {
  indpide; "rembemblodity: if-indphoximosof(int framibh, inn sharedith; {
    indpide; "rembemblodity: if-indphoximosof(int framibh, sharedith; fromthh, 0);
    int whit; nearlib; "rembemblo; "rembemblo; "rembemblo; "rembemblo; "remp = new(nearbit); indpide; "remp = new(nearbit); 
                                          rec) (
(int i=0; ixirec->getShared(); i++) (

!f(irec->getChk() == toChk) (

idxobortsend = fmVtil->exists in IDML(fmMesh, localIdx, toChk, l);
           Owners coord[2];
FEM Nesh_dataP[snleash, FEM NOOE, fmAdaptAlga->coord_atr, coord, localIdx, 1, FEM_DOUBLE, 2];
return did of new doubleThes(coord[1));
return did of new doubleThes(coord[1));
```



A Typical ParFUM Program



Why should I avoid multiparadigm programming?

You can only program in languages you know

MPI is safe and popular

You need modularity

Language choice is limited by the underlying RTS

Productivity

Productivity

Productivity



Because it is a multiparadigm program, ParFUM is:

- Easier to develop and easier to understand
- More extensible and flexible
- Still easy to use by MPI programmers

Charm++ is a great platform for multiparadigm programming, and I encourage you to try it out.

Multiparadigm Parallel Programming with Charm++, Featuring ParFUM as a case study

5th Annual Workshop on Charm++ and its Applications

Aaron Becker abecker3@uiuc.edu UIUC 18 April 2007



