Charm++ Tutorial

Presented by Eric Bohm

Outline

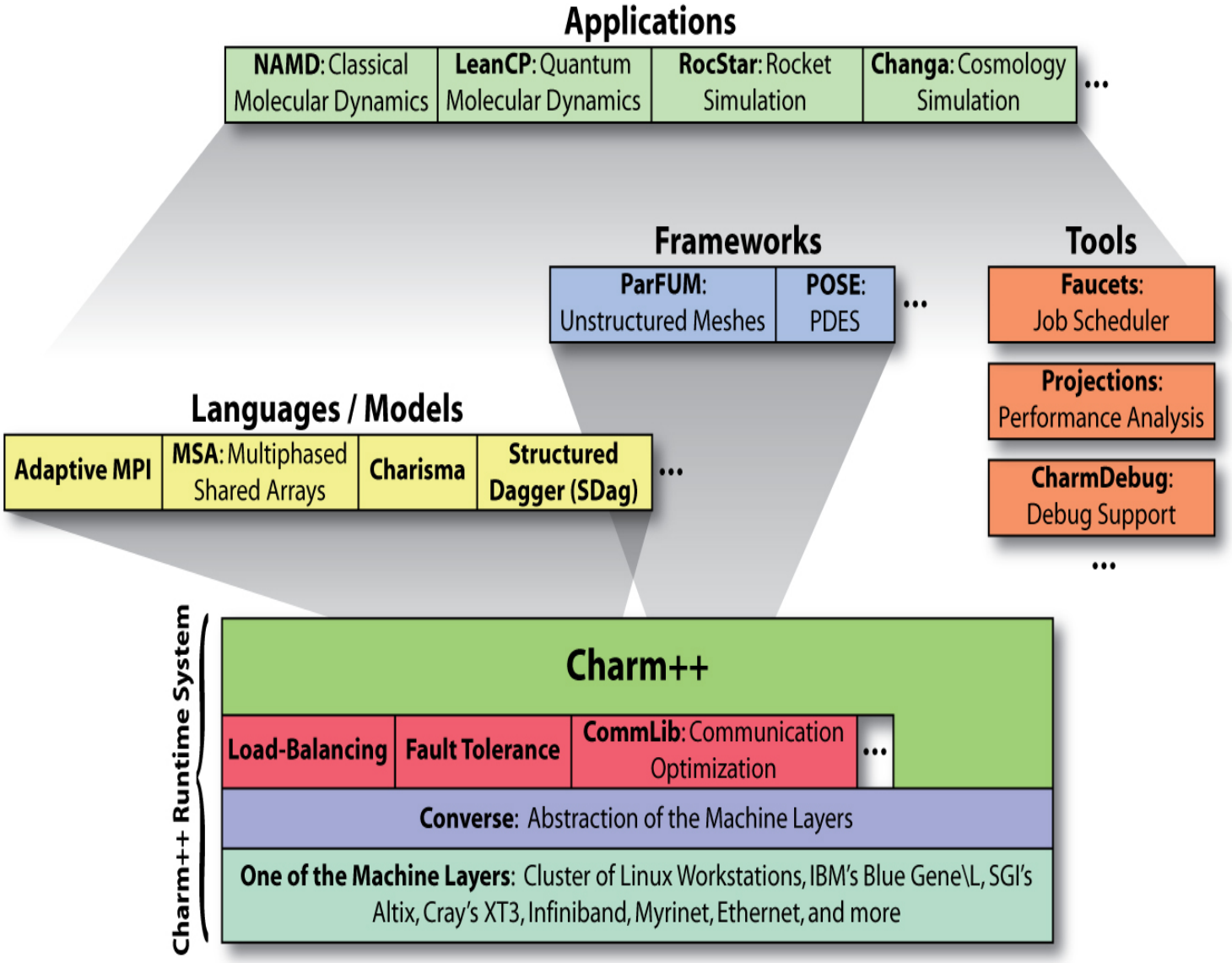
- Basics
  - Introduction
  - Charm++ Objects
  - Chare Arrays
  - Chare Collectives
  - SDAG
  - Example
- Intermission
- Advanced
  - Prioritized Messaging
  - Interface file tricks
    - Initialization
    - Entry Method Tags
  - Groups & Node Groups
  - Threads

Expectations

- Introduction to Charm++
  - Assumes parallel programming aware audience
  - Assume C++ aware audience
  - AMPI not covered
- Goals
  - What Charm++ is
  - How it can help
  - How to write a basic charm program
  - Provide awareness of advanced features

What Charm++ Is Not

- Not Magic Pixie Dust
  - Runtime system exists to help you
  - Decisions and customizations are necessary in proportion to the complexity of your application
- Not a language
  - Platform independent library with a semantic
  - Works for C, C++, Fortran (not covered in this tutorial)
- Not a Compiler
- Not SPMD Model
- Not Processor Centric Model
  - Decompose to individually addressable medium grain tasks
- Not A Thread Model
  - They are available if you want to inflict them on your code
- Not Bulk Synchronous

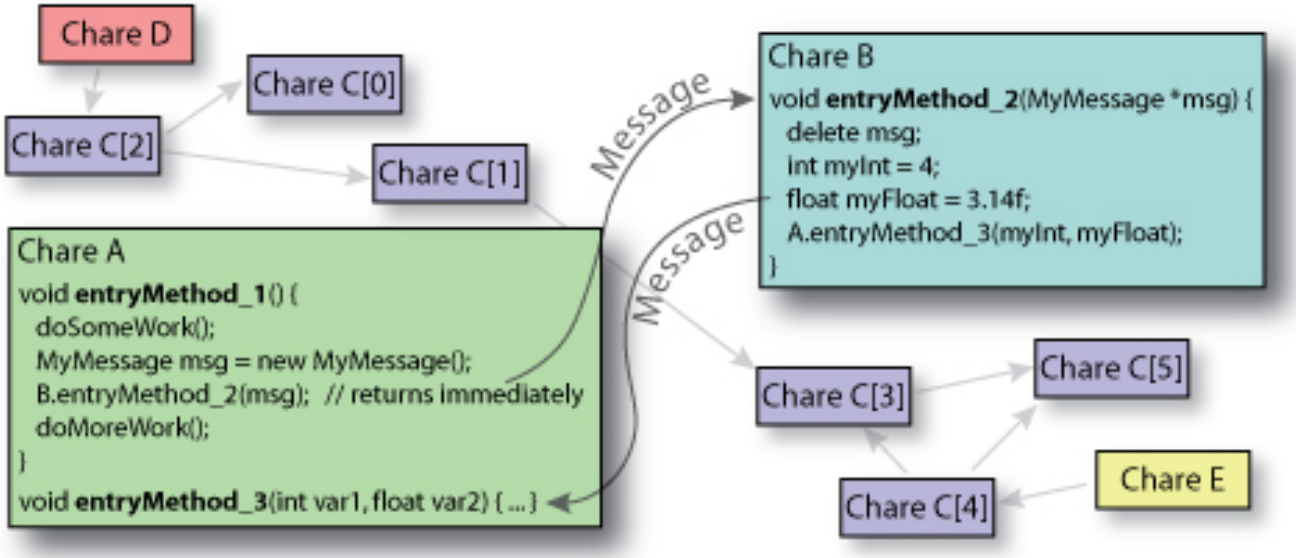

Applications
NAMD: Classical Molecular Dynamics
LeanCP: Quantum Molecular Dynamics
RocStar: Rocket Simulation
Changa: Cosmology Simulation
...
Frameworks
ParFUM: Unstructured Meshes
POSE: PDES
...
Tools
Faucets: Job Scheduler
Projections: Performance Analysis
CharmDebug: Debug Support
...
Languages / Models
Adaptive MPI
MSA: Multiphased Shared Arrays
Charisma
Structured Dagger (SDag)
...
Charm++ Runtime System
Charm++
Load-Balancing
Fault Tolerance
CommLib: Communication Optimization
...
Converse: Abstraction of the Machine Layers
One of the Machine Layers: Cluster of Linux Workstations, IBM's Blue Gene\L, SGI's Altix, Cray's XT3, Infiniband, Myrinet, Ethernet, and more

The Charm++ Model

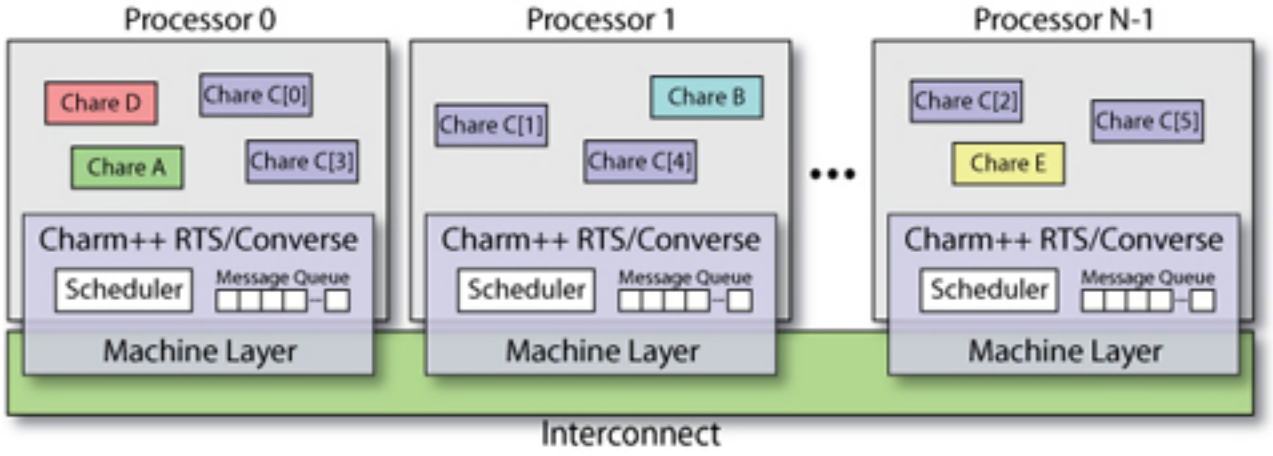
- Parallel objects (chares) communicate via asynchronous method invocations (entry methods).
- The runtime system maps chares onto processors and schedules execution of entry methods.
- Similar to Active Messages or Actors

User View vs. System View

User View:



System View:



Architecures

- Runs on:
  - Any machine with MPI installation
  - Clusters with Ethernet (UDP/TCP)
  - Clusters with Infiniband
  - Clusters with accelerators (GPU/CELL)
  - Windows
  - …
- To install
  - “./build”

Portability

- Cray XT (3|4|5)
  - Cray XT6 in development
- BlueGene (L|P)
  - BG/Q in development
- BlueWaters
  - LAPI
  - PAMI in development
- SGI/Altix

Clusters

X86, X86_64, Itanium

MPI, UDP, TCP, LAPI, Infiniband, Myrinet, Elan, SHMEM

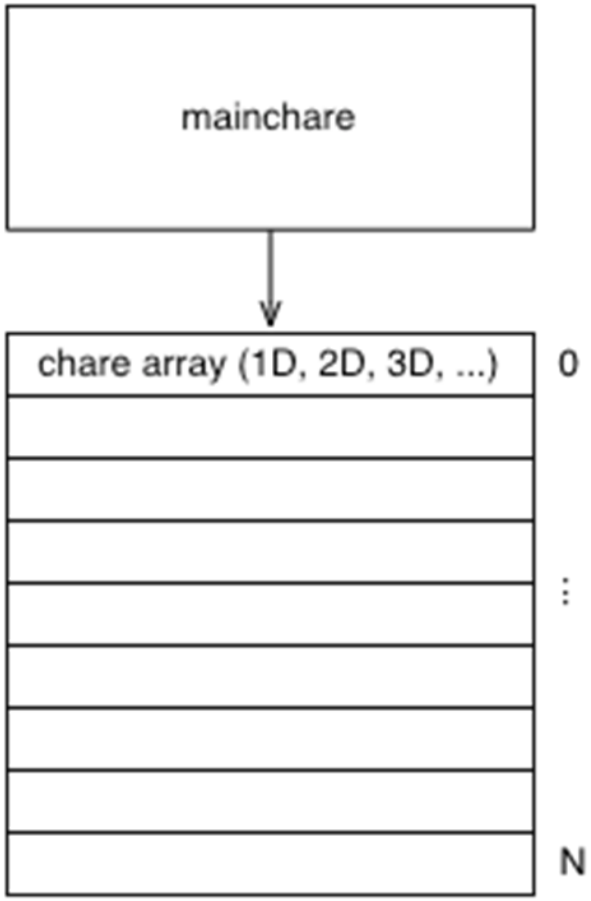
Accelerators

Cell

GPGPU

Charm++ Objects

- A “chare” is a C++ object with methods that can be remotely invoked
- The “mainchare” is the chare where the execution starts in the program
- A “chare array” is a collection of chares of the same type
- Typically the mainchare will spawn a chare array of workers



Charm++ File Structure

- The C++ objects (whether they are chares or not)
  - Reside in regular .h and .cpp files
- Chare objects, messages and entry methods (methods that can be called asynchronously and remotely)
  - Are defined in a .ci (Charm interface) file
  - And are implemented in the .cpp file

Hello World: .ci file

- .ci: Charm Interface
- Defines which type of chares are present in the application
  - At least a *mainchare* must be set
- Each definition is inside a *module*
  - Modules can be included in other modules

```
mainmodule hello {

 mainchare Main {
  entry Main(CkArgMsg* msg);
 };

};
```

Hello World: the code

main.h

```
#include ”hello.decl.h”

class Main : public CBase_Main {

  public:
    Main(CkArgMsg* msg);
    Main(CkMigrateMessage* msg);
};
```

main.C

```
#include "main.h"

// Entry point of Charm++ application
Main::Main(CkArgMsg* msg) {

  CkPrintf("Hello World!\n");

  CkExit();
}

Main::Main(CkMigrateMessage* msg) { }

#include "hello.def.h"
```

CkArgMsg in the Main::Main Method

- Defined in charm++
- struct CkArgMsg{

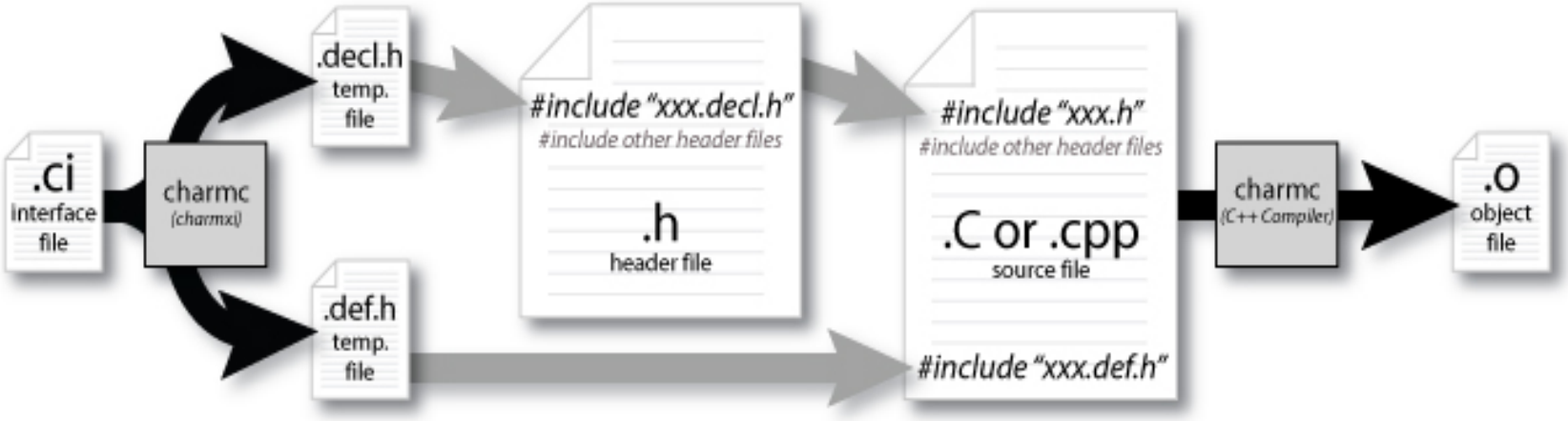
  int argc;

  char **argv;

  }

Compilation Process

- charmc hello.ci
- charmc –o main.o main.C (compile)
- charmc –language charm++ -o pgm main.o (link)

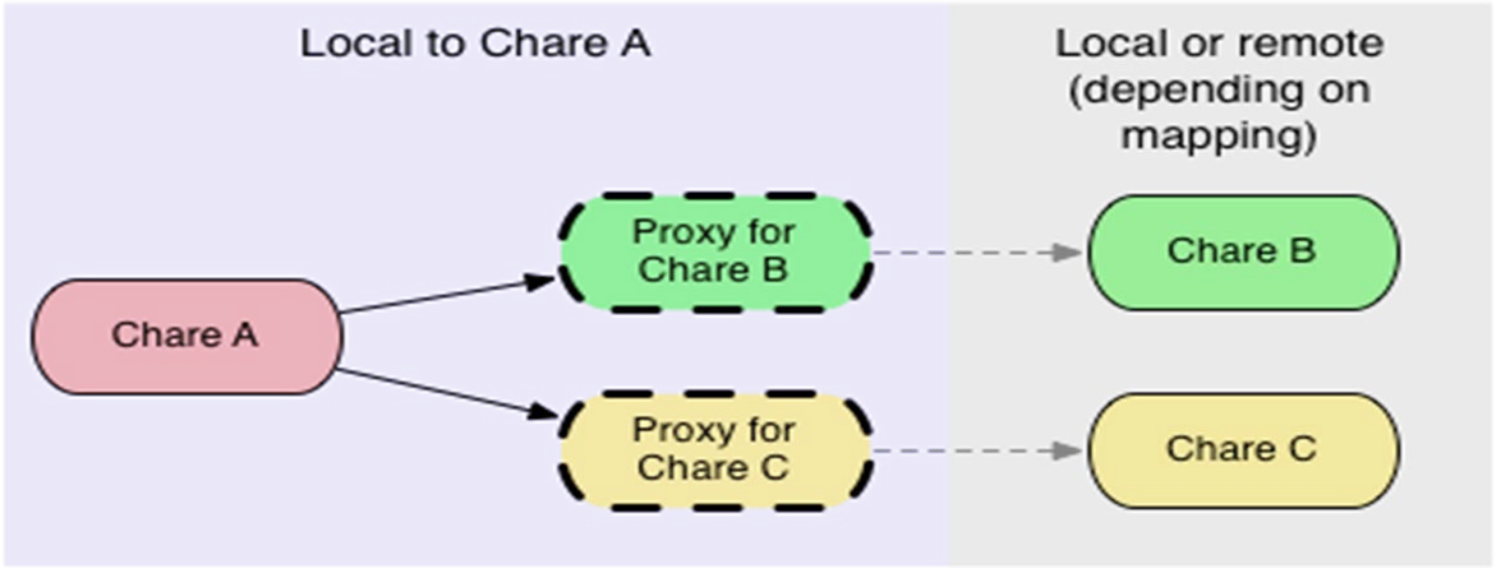


Execution

- ./charmrun +p4 ./pgm
  - Or specific queueing system
- Output:
  - Hello World!
- Not a parallel code :(
  - Solution: create other chares, all of them saying "Hello World"

How to Communicate?

- Chares spread across multiple processors
  - It is not possible to directly invoke methods
- Use of Proxies – lightweight handles to potentially remote chares



The Proxy

- A Proxy class is generated for every chare
  - For example, Cproxy_Main is the proxy generated for the class Main
  - Proxies know where a chare is inside the system
  - Methods invoked on a Proxy pack the input parameters, and send them to the processor where the chare is. The real method will be invoked on the destination processor.
- Given a Proxy p, it is possible to call the method
  - p.method(msg)

A Slightly More Complex Hello World

- Program's asynchronous flow
  - Mainchare sends message to Hello object
  - Hello object prints "Hello World!"
  - Hello object sends message back to the mainchare
  - Mainchare quits the application

Code

hello.ci

```
mainmodule hello {
  readonly CProxy_Main mainProxy;

  mainchare Main {
    entry Main(CkArgMsg*);
    entry void end(void);
  };

  chare Hello {
    entry Hello();
    entry void PrintHello(void);
  }
};
```

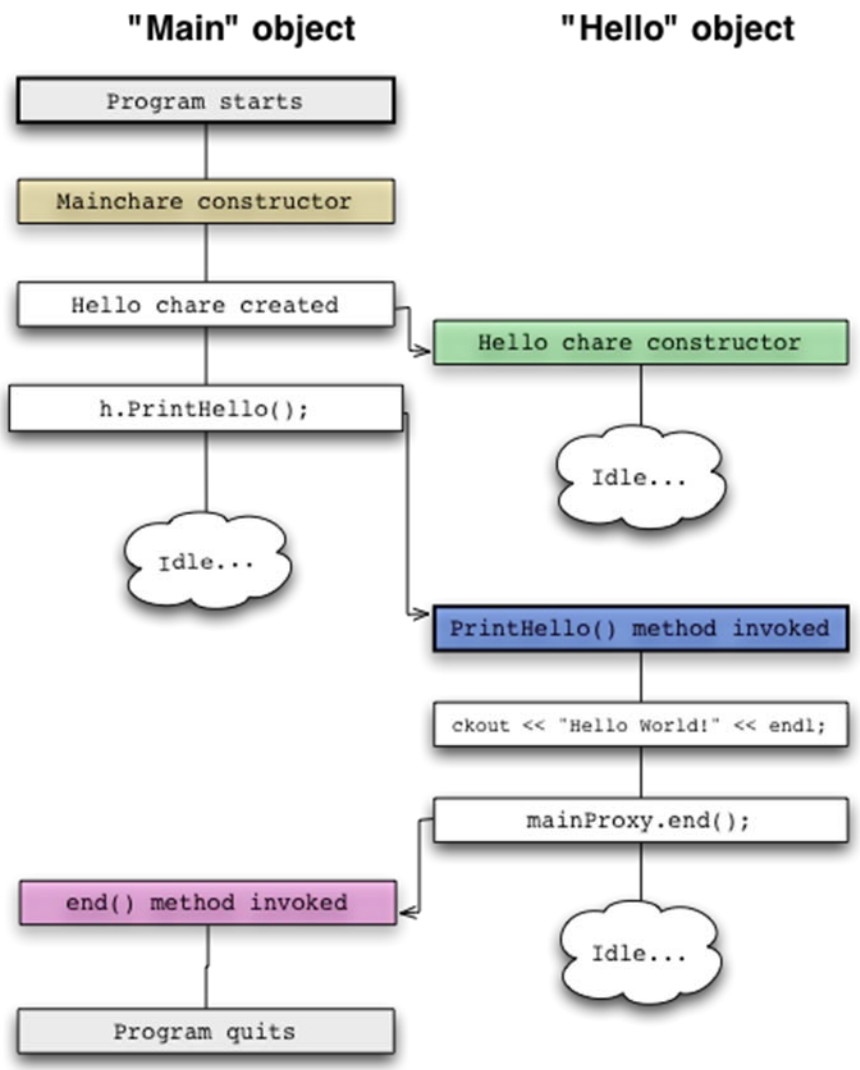
hello.cpp

```
#include "hello.decl.h"

/*readonly*/ CProxy_Main mainProxy;

class Main : public Chare {
public:
  Main(CkArgMsg* m) {
    delete m;
    mainProxy = thishandle;

    CProxy_Hello h = CProxy_Hello::ckNew();
    h.PrintHello();
  }

  void end() {
    CkExit();
  }
};

class Hello : public CBase_Hello {
public:
  Hello() {}

  void PrintHello(void) {
    ckout << "Hello World!" << endl;
    mainProxy.end();
  }
};

#include "hello.def.h"
```

"*readonly*" Variables

- Defines a global variable
  - Every PE has its value
- Can be **set only in the *mainchare***!

Workflow of Hello World




```
#include "hello.decl.h"

/*readonly*/ CProxy_Main mainProxy;

class Main : public Chare {
public:
  Main(CkArgMsg* m) {
    delete m;
    mainProxy = thishandle;

    CProxy_Hello h = CProxy_Hello::ckNew();
    h.PrintHello();
  }

  void end() {
    CkExit();
  }
};

class Hello : public CBase_Hello {
public:
  Hello() {}

  void PrintHello(void) {
    ckout << "Hello World!" << endl;
    mainProxy.end();
  }
};

#include "hello.def.h"
```

Limitations of Plain Proxies

- In a large program, keeping track of all the proxies is difficult
- A simple proxy doesn't tell you anything about the chare other than its type.
- Managing collective operations like broadcast and reduce is complicated.

Chare Arrays

- Arrays organize chares into indexed collections.
- There is a single name for the whole collection
- Each chare in the array has a proxy for the other array elements, accessible using simple syntax
  - `sampleArray[i] // i'th proxy`

Array Dimensions

- Anything can be used as array indices
  - integers
  - Tuples (e.g., 2D, 3D array)
  - bit vectors
  - user-defined types

Array Elements Mapping

- Automatically by the runtime system
- Programmer could control the mapping of array elements to PEs.
  - Round-robin, block-cyclic, etc
  - User defined mapping

Broadcasts

- Simple way to invoke the same entry method on each array element.
- Example: A 1D array “Cproxy_MyArray arr”
  - arr[3].method(): a point-to-point message to element 3.
  - arr.method(): a ***broadcast*** message to every elements

Hello World: Array Version

hello.ci

```
mainmodule hello {

  readonly CProxy_Main mainProxy;
  readonly int numElements;

  mainchare Main {
    entry Main(CkArgMsg* msg);
    entry void done();
  };

  array [1D] Hello {
    entry Hello();
    entry void sayHi(int);
  };

};
```

- entry *void* sayHi(*int*)
  - Not meaningful to return a value
  - Parameter marshalling: runtime system will automatically pack arguments into a message or unpack the message into arguments

Hello World: Main Code

main.h

```
#include "hello.decl.h"

class Main : public CBase_Main {

 public:

  Main(CkArgMsg* msg);
  Main(CkMigrateMessage* msg) {}

  void done();
};
```

main.c

```
#include "main.h"

/* readonly */ CProxy_Main mainProxy;
/* readonly */ int numElements;

Main::Main(CkArgMsg* msg) {
  numElements = 5; // Default numElements to 5

  if (msg->argc > 1) numElements = atoi(msg->argv[1]);
  // We are done with msg so delete it.
  delete msg;

  CkPrintf("Running \"Hello World\" with %d elements "
       "using %d processors.\n", numElements, CkNumPes());

  mainProxy = thisProxy;

  CProxy_Hello helloArray = CProxy_Hello::ckNew(numElements);

  helloArray[0].sayHi(-1);
}

void Main::done() {
  CkExit();
}
```

Hello World: Array Code

hello.h

```
#include "hello.decl.h"

class Hello : public CBase_Hello {

 public:

  Hello();
  Hello(CkMigrateMessage *msg) {}

  void sayHi(int from);
};
```

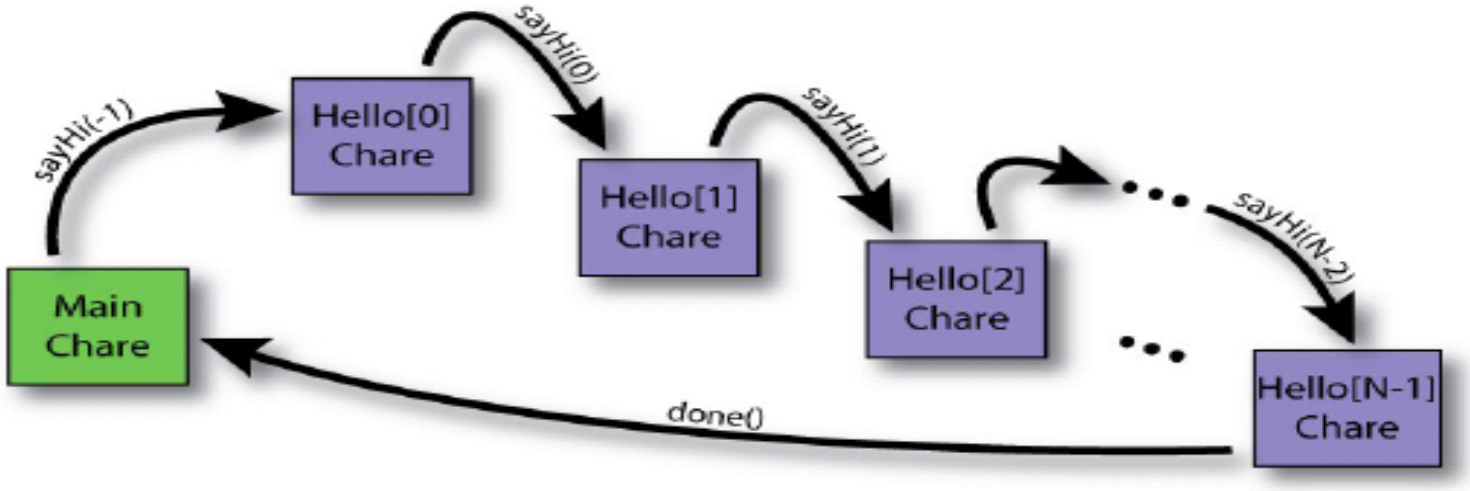
hello.C

```
#include "hello.h"

extern /* readonly */ CProxy_Main mainProxy;
extern /* readonly */ int numElements;

Hello::Hello() { }

void Hello ::sayHi(int from) {

  CkPrintf("\"Hello\" from Hello chare # %d on "
           "processor %d (told by %d).\n",
           thisIndex, CkMyPe(), from);

  if (thisIndex < (numElements - 1))
    thisProxy[thisIndex + 1].sayHi(thisIndex);
  else
    mainProxy.done();
}

#include "hello.def.h"
```

Result

$ ./charmrun +p3 ./hello 10

Running “Hello World” with 10 elements using 3 processors.
“Hello” from Hello chare #0 on processor 0 (told by -1)
“Hello” from Hello chare #1 on processor 0 (told by 0)
“Hello” from Hello chare #2 on processor 0 (told by 1)
“Hello” from Hello chare #3 on processor 0 (told by 2)
“Hello” from Hello chare #4 on processor 1 (told by 3)
“Hello” from Hello chare #5 on processor 1 (told by 4)
“Hello” from Hello chare #6 on processor 1 (told by 5)
“Hello” from Hello chare #7 on processor 2 (told by 6)
“Hello” from Hello chare #8 on processor 2 (told by 7)
“Hello” from Hello chare #9 on processor 2 (told by 8)

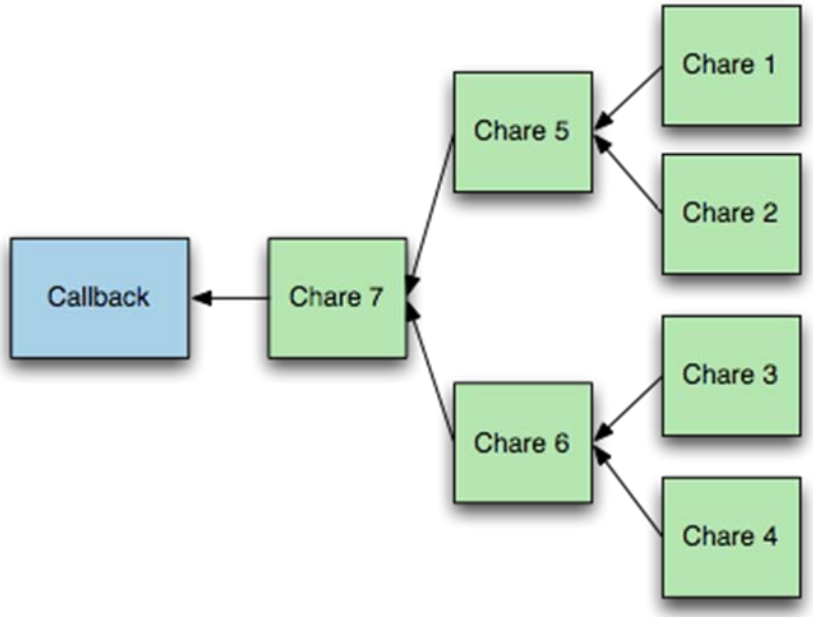


Reduction (1)

- Every chare element will contribute its portion of data to someone, and data are combined through a particular *op.*
- Naïve way:
  - Use a “master” to count how many messages need to be received.
  - Potential bottleneck on the “master”

Reduction (2)

- Runtime system builds reduction tree
- User specifies reduction *op*
- At root of tree, a callback is performed on a specified chare



Reduction in Charm++

- No global flow of control, so each chare must contribute data independently using `contribute(…)`.
  - void contribute(int nBytes, const void *data, CkReduction::reducerType type):
- A *user callback* (created using `CkCallback`) is invoked when the reduction is complete.

Reduction *Ops* (CkReduction::reducerType)

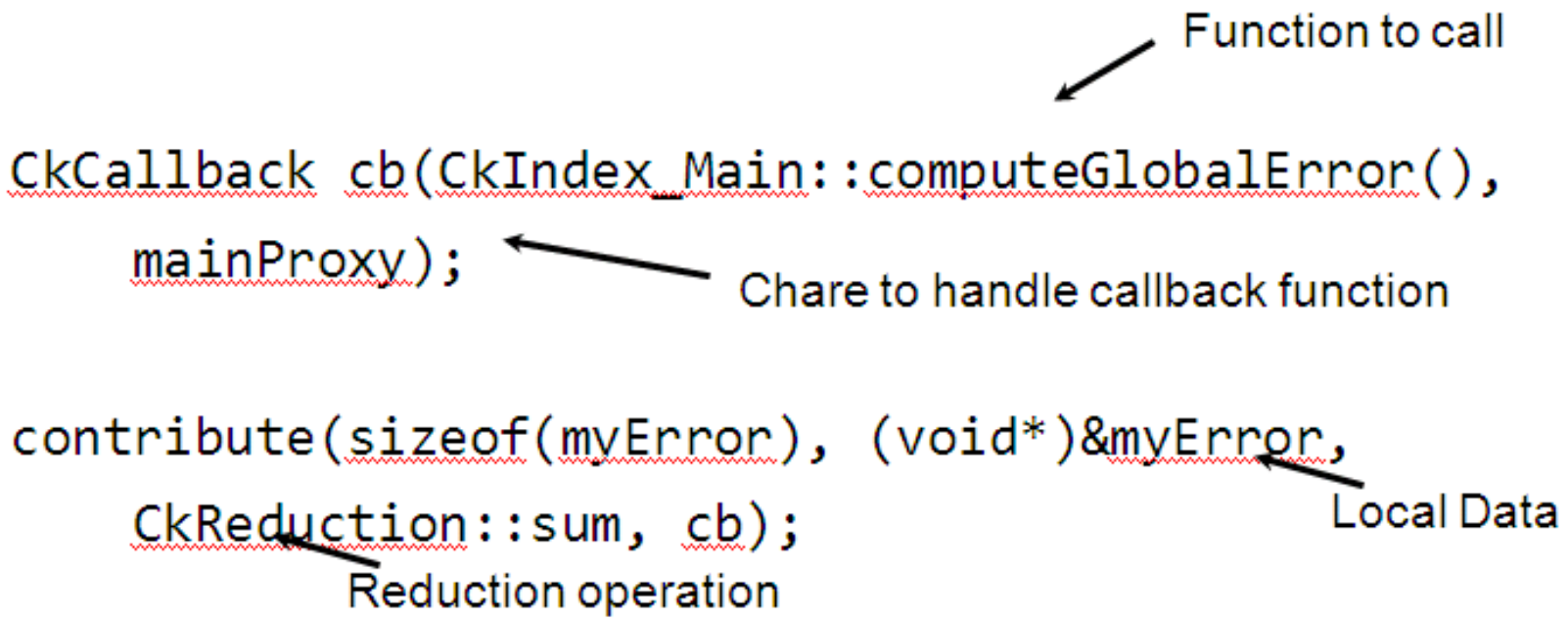
- Predefined:
  - Arithmetic (int, float, double)
    - CkReduction::sum_int, …
    - CkReduction::product_int, …
    - CkReduction::max_int, …
    - CkReduction::min_int, …
  - Logic:
    - CkReduction::logical_and, logic_or
    - CkReduction::bitvec_and, bitvec_or
  - Gather:
    - CkReduction::set, concat
  - Misc:
    - CkReduction::random
- Defined by the user

Callback: where reductions go?

- CkCallback(CkCallbackFn fn, void *param)
  - void myCallbackFn(void *param, void *msg)
- CkCallback(int ep, const CkChareID &id)
  - ep=CkIndex_ChareName::EntryMethod(parameters)
- CkCallback(int ep, const CkArrayID &id)
  - A Cproxy_MyArray may substitute CkArrayID
    - The callback will be called on all array elements
- CkCallback(int ep, const CkArrayIndex &idx, const CkArrayID &id)
  - The callback will only be called on element[idx]
- CkCallback(CkCallback::ignore)

Example

- Sum local error estimators to determine global error

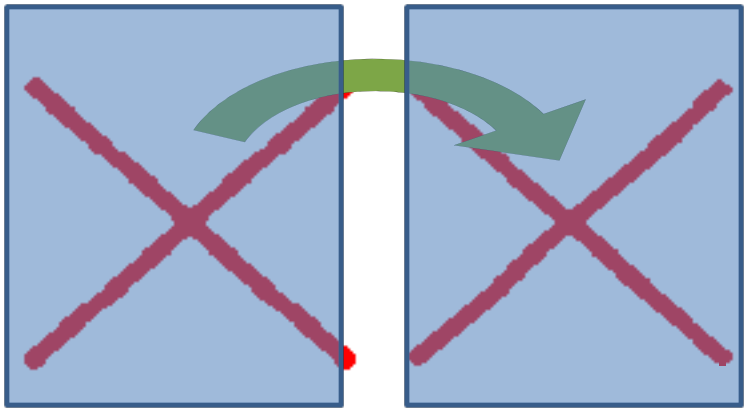


SDAG JACOBI Example

- Introduce SDAG
- Using 5 point stencil

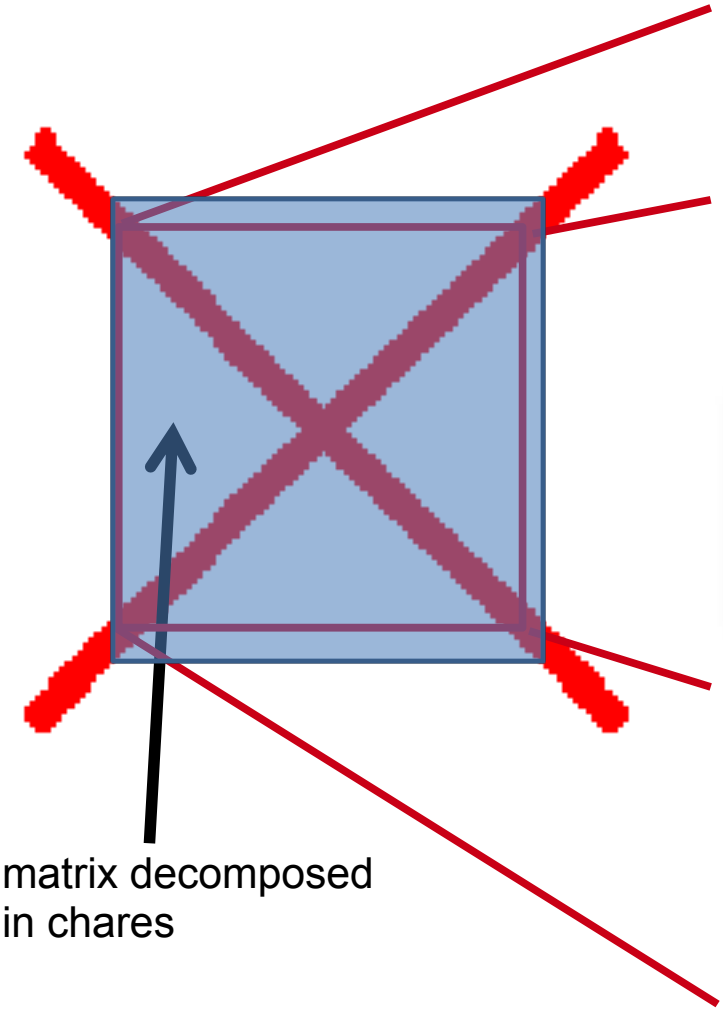
Example: Jacobi 2D

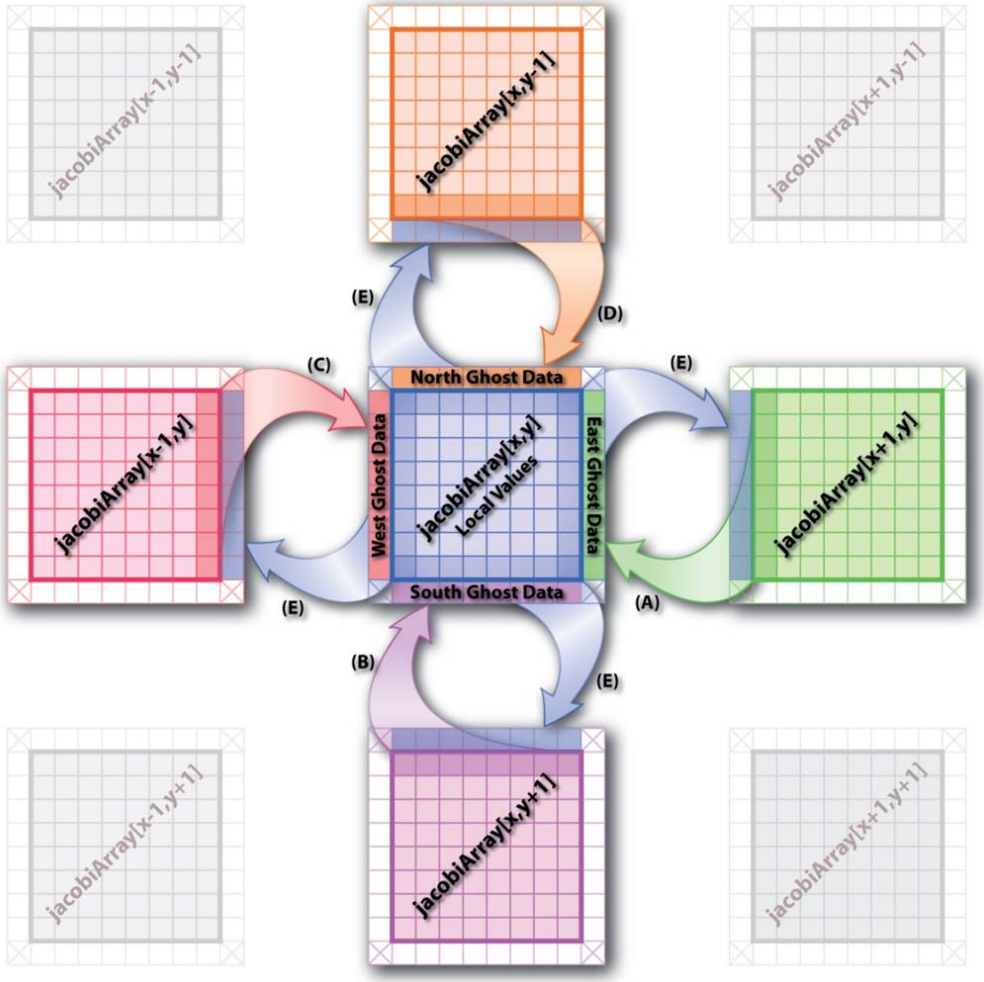
- Use two interchangeable matrices



```
do {
   update_matrix();
   maxDiff = max(abs (A - B));
} while (maxDiff > DELTA)

update_matrix() {
   foreach i,j {
      B[i,j] = (A[i,j] + A[i+1,j] + A[i-1,j] + A[i,j+1] + A[i,j-1]) / 5;
   }
   swap (A, B);
}
```

Jacobi in parallel





Jacobi: the code

```
Main(CkArgMsg* m) { // initialize everything
    array = CProxy_Jacobi::ckNew(num_chare_x, num_chare_y);
    array.begin_iteration();
}
 void report(CkReductionMsg *msg) { // Each worker reports back to here when it completes an iteration
    iterations++;
    maxdifference=((double *) msg->getData())[0];
    delete msg;
    if ( maxdifference – THRESHHOLD<0)  {
                    CkPrintf("Difference %.10g Satisfied Threshhold %.10g in %d Iterations\n",
 maxdifference,THRESHHOLD,iterations);
                    done(true); }
    else {array.begin_iteration();}
  }
void Jacobi::begin_iteration(void) {
    iterations++;
    if(!leftBound)
      {             double *leftGhost =  new double[blockDimY];
                    for(int j=0; j<blockDimY; ++j)
                      leftGhost[j] = temperature[index(1, j+1)];
                    thisProxy(thisIndex.x-1, thisIndex.y)
                      .processGhosts( RIGHT, blockDimY, leftGhost);
                    delete [] leftGhost; }
…
}
void processGhosts(int dir, int size, double gh[]) {
    switch(dir) {
    case LEFT:
      for(int j=0; j<size; ++j) emperature[index(0, j+1)] = gh[j];
 ...
    if(++imsg==numExpected) check_and_compute();     }
  }
 void check_and_compute() {
    imsg=0;
    compute_kernel();
    contribute(sizeof(double), &maxdifference, CkReduction::max_double, CkCallback(CkIndex_Main::report(NULL), mainProxy));
```

```
mainmodule jacobi2d {

  readonly CProxy_Main mainProxy;
  readonly int arrayDimX; readonly int arrayDimY;
  readonly int blockDimX; readonly int blockDimY;
  readonly int num_chare_x; readonly int num_chare_y;
  readonly int maxiterations;

  mainchare Main {
    entry Main(CkArgMsg *m);
    entry void report(CkReductionMsg *m);
  };

  array [2D] Jacobi {
    entry Jacobi(void);
    entry void begin_iteration(void);
    entry void processGhosts(int dir, int size, double ghosts[size]);
    };
};
```

Remove Barrier

- More efficient
- Problem!
  - Potential Race Condition
  - May receive neighbor update for next iteration
- Solution
  - Send iteration counter
  - Buffer (and count for next iter) messages until ready

We can do better using SDAG

- Structured DAGger
  - Directed Acyclic Graph (DAG)
- Express event sequencing and dependency
- Automate Message buffering
- Automate Message counting
- Express independence for overlap
- Differentiate between parallel and sequential blocks
- Negligible overhead

Structured Dagger Constructs

- when <method list> {code}
  - Do not continue until method is called
  - Internally generates flags, checks, etc.
- atomic {code}
  - Call ordinary sequential C++ code
- if/else/for/while
  - C-like control flow
- overlap {code1 code2 ...}
  - Execute code segments in parallel
- forall
  - “Parallel Do”
  - Like a parameterized overlap

Reinvent Jacob2d in SDAG

- Code walkthrough
- Task 1
  - Convert to SDAG
  - Add _sdag directives
  - Add sdag control entry method
  - Make distinction between receiving and processing ghosts
  - Use SDAG iteration and message counting
  - Remove barrier

Jacob2d to 3d in SDAG

- Hands on project homework
- Task 2
  - Convert to 3D 7point stencil
  - Add “front” “back” neighbors and blocksizes
  - Revise numExpected calculation
  - Add FRONT BACK ghost cases
  - Add frontBound backBound,kStart, kFinish
  - Extend index(), k dimension to init + compute
- Is there a need to change the SDAG code?
- Answer can be found in Charm++ distribution

Intermission

Advanced Messaging

Prioritized Execution

- **Charm++ scheduler**
  - **Default - FIFO (oldest message)**
- **Prioritized execution**
  - **If several messages available, Charm will process the messages in the order of their priorities**
- **Very useful for speculative work, ordering timestamps, etc...**

Priority Classes

- **Charm++ scheduler has three queues: high, default, and low**
- **As signed integer priorities:**
  - **High -MAXINT to -1**
  - **Default 0**
  - **Low 1 to +MAXINT**
- **As unsigned bitvector priorities:**
  - **0x0000 Highest priority -- 0x7FFF**
  - **0x8000 Default priority**
  - **0x8001 -- 0xFFFF Lowest priority**

Prioritized Messages

- **Number of priority bits passed during message allocation**

```
FooMsg * msg = new (size, nbits) FooMsg;
```

  - **Priorities stored at the end of messages**

- **Signed integer priorities**

```
*CkPriorityPtr(msg)=-1;
CkSetQueueing(msg, CK_QUEUEING_IFIFO);
```

- **Unsigned bitvector priorities**

```
CkPriorityPtr(msg)[0]=0x7fffffff;
CkSetQueueing(msg, CK_QUEUEING_BFIFO);
```

Prioritized Marshalled Messages

- **Pass "CkEntryOptions" as last parameter**
- **For signed integer priorities:**

```
CkEntryOptions opts;
opts.setPriority(-1);
fooProxy.bar(x,y,opts);
```

- **For bitvector priorities:**

```
CkEntryOptions opts;
unsigned int prio[2]={0x7FFFFFFF,0xFFFFFFFF};
opts.setPriority(64,prio);
fooProxy.bar(x,y,opts);
```

Advanced Message Features

- **Nokeep (Read-only) messages**
  - **Entry method agrees not to modify or delete the message**
  - **Avoids message copy for broadcasts, saving time**
- **Inline messages**
  - **Direct method invocation if on local processor**
- **Expedited messages**
  - **Message do not go through the charm++ scheduler (ignore any Charm++ priorities)**
- **Immediate messages**
  - **Entries are executed in an interrupt or the communication thread**
  - **Very fast, but tough to get right**
  - **Immediate messages only currently work for NodeGroups and Group (non-smp)**

Read-Only, Expedited, Immediate

- **All declared in the .ci file**

```
  {
   entry [nokeep] void foo_readonly(Msg *);
   entry [inline] void foo_inl(Msg *);
   entry [expedited] void foo_exp(Msg *);
   entry [immediate] void foo_imm(Msg *);
   ...
};
```

Interface File Example

```
mainmodule hello {
  include “myType.h”

  initnode void myNodeInit();
  initproc void myInit();

  mainchare mymain {
    entry mymain(CkArgMsg *m);
  };

  array[1D] foo {
    entry foo(int problemNo);
    entry void bar1(int x);
    entry void bar2(myType x);
  };
};
```

Include and Initcall

- **Include**
  - **Include an external header files**
- **Initcall**
  - **User plugging code to be invoked in Charm++'s startup phase**
- **Initnode**
  - **Called once on every node**
- **Initproc**
  - **Called once on every processor**
- **Initnode calls are called before Initproc calls**

Entry Attributes

- **Threaded**
  - **Function is invoked in a CthThread**
- **Sync**
  - **Blocking methods, can return values as a message**
  - **Caller must be a thread**
- **Exclusive**
  - **For Node Group**
  - **Do not execute while other exclusive entry methods of its node group are executing in the same node**
- **Notrace**
  - **Invisible to trace projections**
  - entry [notrace] void recvMsg(multicastGrpMsg *m);

Entry Attributes 2

- **Local**
  - **Local function call, traced like an entry method**
- **Python**
  - **Callable by python scripts**
- **Exclusive**
  - **For Node Group**
  - **Do not execute while other exclusive entry methods of its node group are executing in the same node**

Groups/Node Groups

Groups and Node Groups

- **Groups**
  - **Similar to arrays:**
    - **Broadcasts, reductions, indexing**
  - **But not completely like arrays:**
    - **Non-migratable; one per processor**
  - **Exactly one representative on each processor**
    - **Ideally suited for system libraries**
  - **Historically called branch office chares (BOC)**
- **Node Groups**
  - **One per SMP node**

Declarations

- **.ci file**

```
group mygroup {
    entry mygroup(); //Constructor
    entry void foo(foomsg *); //Entry method
 };
 nodegroup mynodegroup {
    entry mynodegroup(); //Constructor
    entry void foo(foomsg *); //Entry method
 };
```

- **C++ file**

```
class mygroup : public Group {
    mygroup() {}
  void foo(foomsg *m) { CkPrintf("Do Nothing");}
};
class mynodegroup : public NodeGroup {
    mynodegroup() {}
  void foo(foomsg *m) { CkPrintf("Do Nothing");}
};
```

Creating and Calling Groups

- **Creation**

```
p = CProxy_mygroup::ckNew();
```

- **Remote invocation**

```
p.foo(msg);   //broadcast
p[1].foo(msg);  //asynchronous
p.foo(msg, npes, pes); // list send
```

- **Direct local access**

```
mygroup *g=p.ckLocalBranch();
g->foo(….); //local invocation
```

  - **Danger: if you migrate, the group stays behind!**

Threads in Charm++

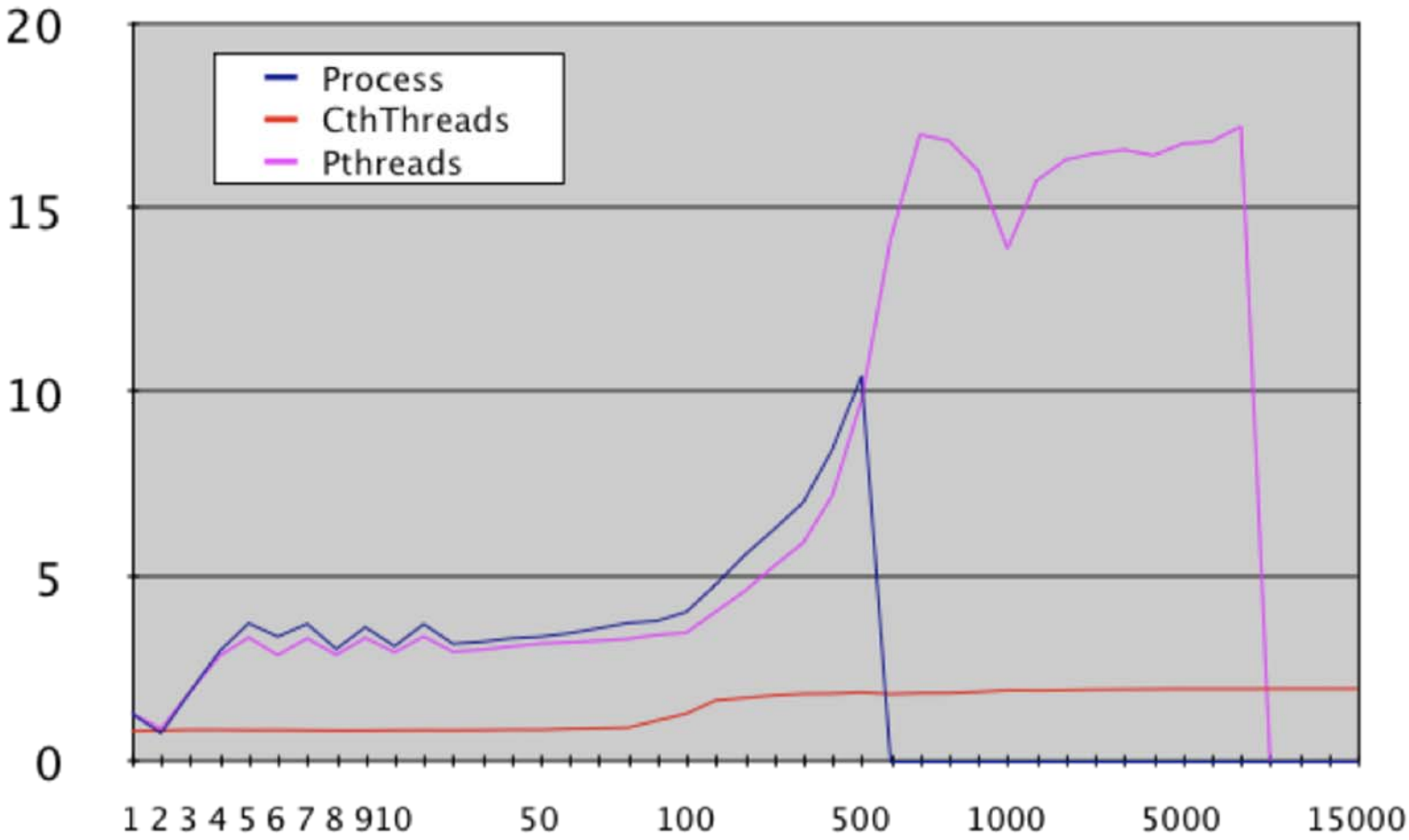
Why use Threads?

- They provide one key feature: blocking
  - Suspend execution (e.g., at message receive)
  - Do something else
  - Resume later (e.g., after message arrives)
- Example: MPI_Recv, MPI_Wait semantics
- Function call interface more convenient than message-passing
  - Regular call/return structure (no CkCallbacks) with complete control flow
  - Allows blocking in middle of deeply nested communication subroutine

Why not use Threads?

- **Slower**
  - **Around 1us context-switching overhead unavoidable**
  - **Creation/deletion perhaps 10us**
- **Migration more difficult**
- **State of thread is scattered through stack, which is maintained by compiler**
- **By contrast, state of object is maintained by users**
- **Thread disadvantages form the motivation to use SDAG**

Context Switch Cost



What are (Converse) Threads?

- **One flow of control (instruction stream)**
  - **Machine Registers & program counter**
  - **Execution stack**
- **Like pthreads (kernel threads)**
- **Only different:**
  - **Implemented at user level (in Converse)**
  - **Scheduled at user level; non-preemptive**
  - **Migratable between nodes**

How do I use Threads?

- **Many options:**
  - **AMPI**
    - **Always uses threads via TCharm library**
  - **Charm++**
    - **[threaded] entry methods run in a thread**
    - **[sync] methods**
  - **Converse**
    - **C routines CthCreate/CthSuspend/CthAwaken**
    - **Everything else is built on these**
    - **Implemented using**
      - **SYSV makecontext/setcontext**
      - **POSIX setjmp/alloca/longjmp**
      - **Assembly code**

How do I use Threads (example)

- **Blocking API routine: find array element**

```
int requestFoo(int src) {
  myObject *obj=...;
  return obj->fooRequest(src)
}
```

- **Send request and suspend**

```
int myObject::fooRequest(int src) {
  proxy[dest].fooNetworkRequest(thisIndex);
  stashed_thread=CthSelf();
  CthSuspend();   // -- blocks until awaken call --
  return stashed_return;
}
```

- **Awaken thread when data arrives**

```
void myObject::fooNetworkResponse(int ret) {
  stashed_return=ret;
  CthAwaken(stashed_thread);
}
```

How do I use Threads (example)

- **Send request, suspend, recv, awaken, return**

```
int myObject::fooRequest(int src) {
  proxy[dest].fooNetworkRequest(thisIndex);
  stashed_thread=CthSelf();
  CthSuspend();

  return stashed_return;
}
```

```
void myObject::fooNetworkResponse(int ret) {
  stashed_return=ret;
  CthAwaken(stashed_thread);
}
```

Thread Migration

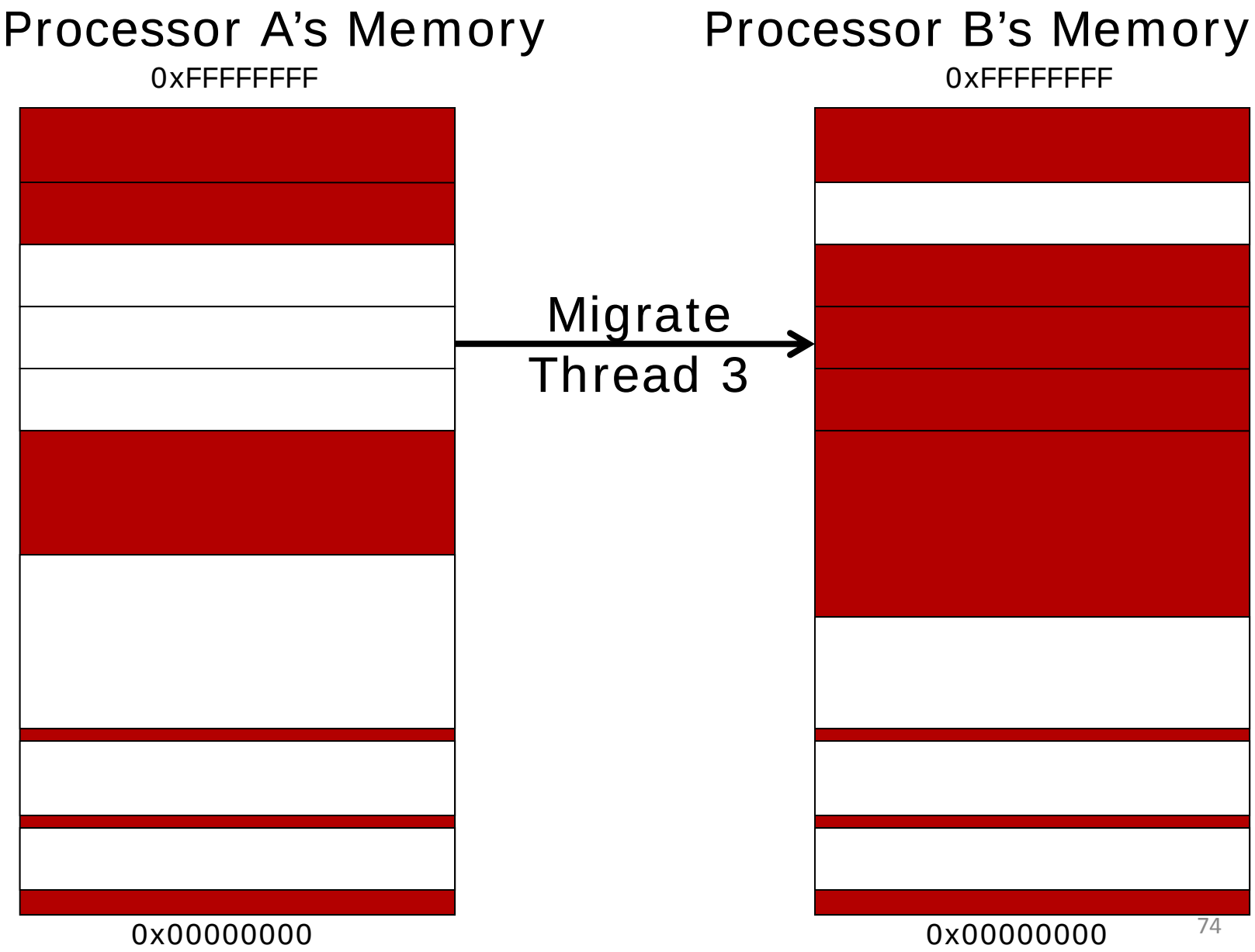
Stack Data

- **The stack is used by the compiler to track function calls and provide temporary storage**
  - **Local Variables**
  - **Subroutine Parameters**
  - **C “alloca” storage**
- **Most of the variables in a typical application are stack data**
- **Stack is allocated by Charm run-time as heap memory (+stacksize)**

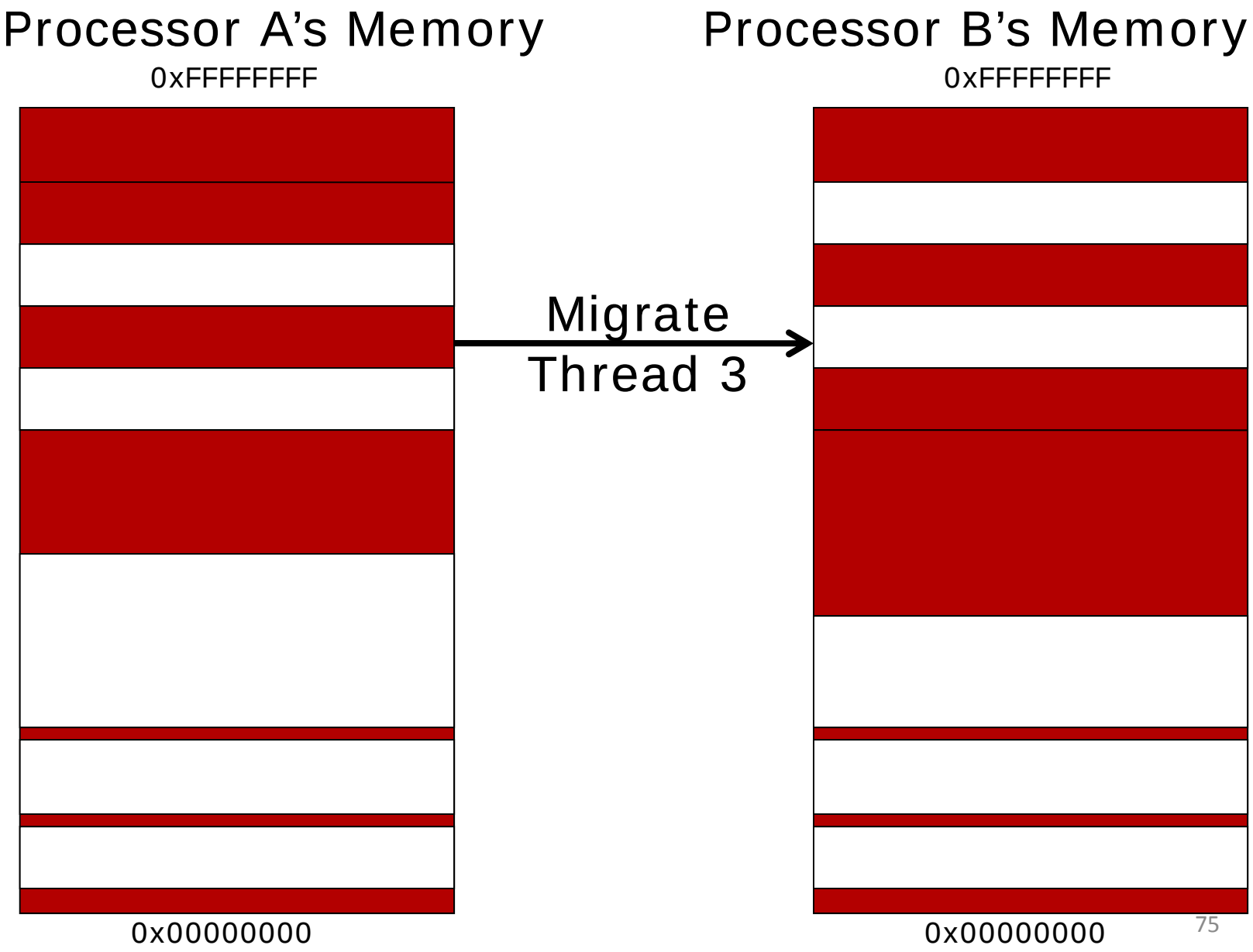
Migrate Stack Data

- **Without compiler support, cannot change stack's address**
  - **Because we can't change stack's interior pointers (return frame pointer, function arguments, etc.)**
- **Existing pointers to addresses in original stack become invalid**
- **Solution: "isomalloc" addresses**
  - **Reserve address space on every processor for every thread stack**
  - **Use *mmap* to scatter stacks in virtual memory efficiently**
  - **Idea comes from PM^2**

Migrate Stack Data



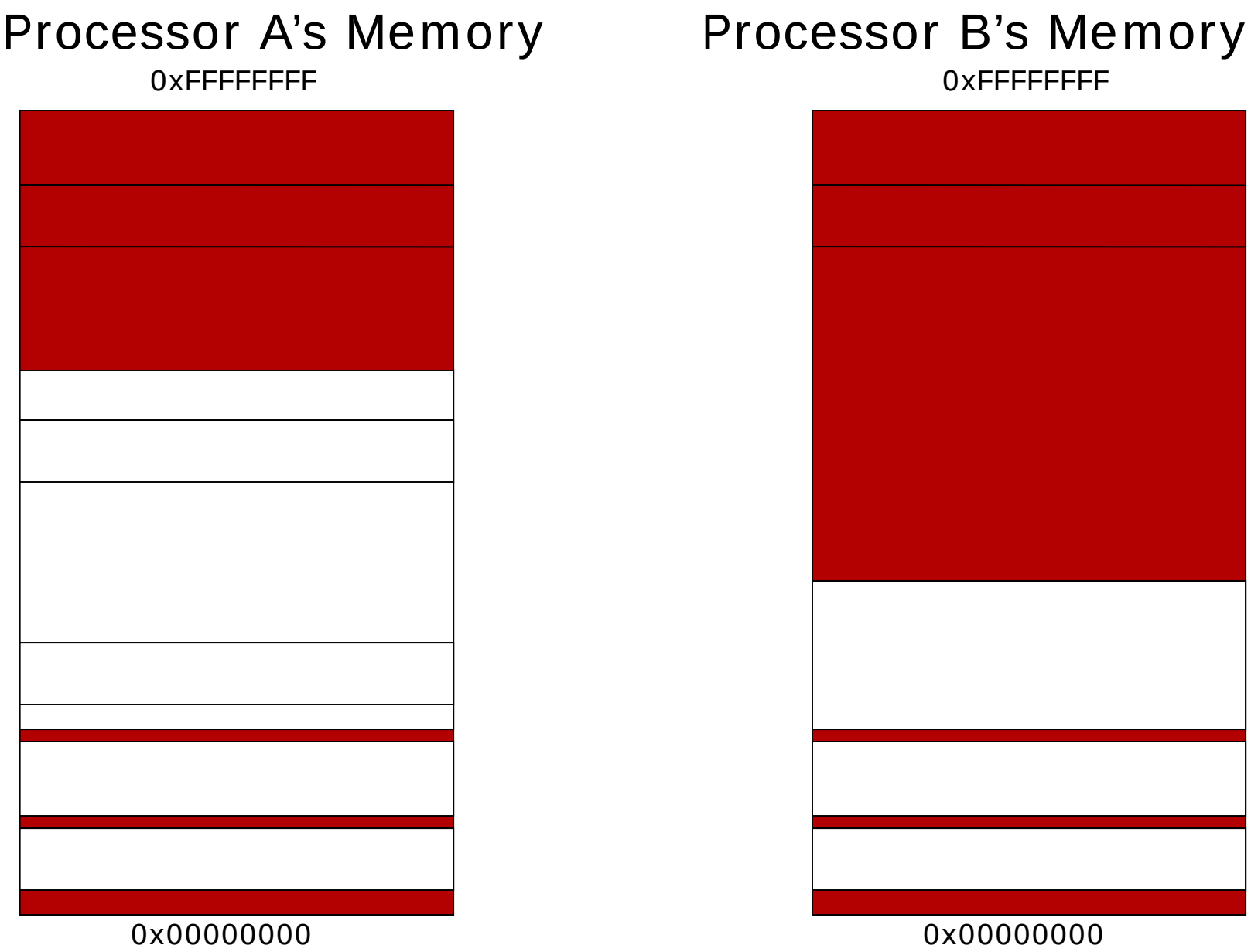
Migrate Stack Data: Isomalloc



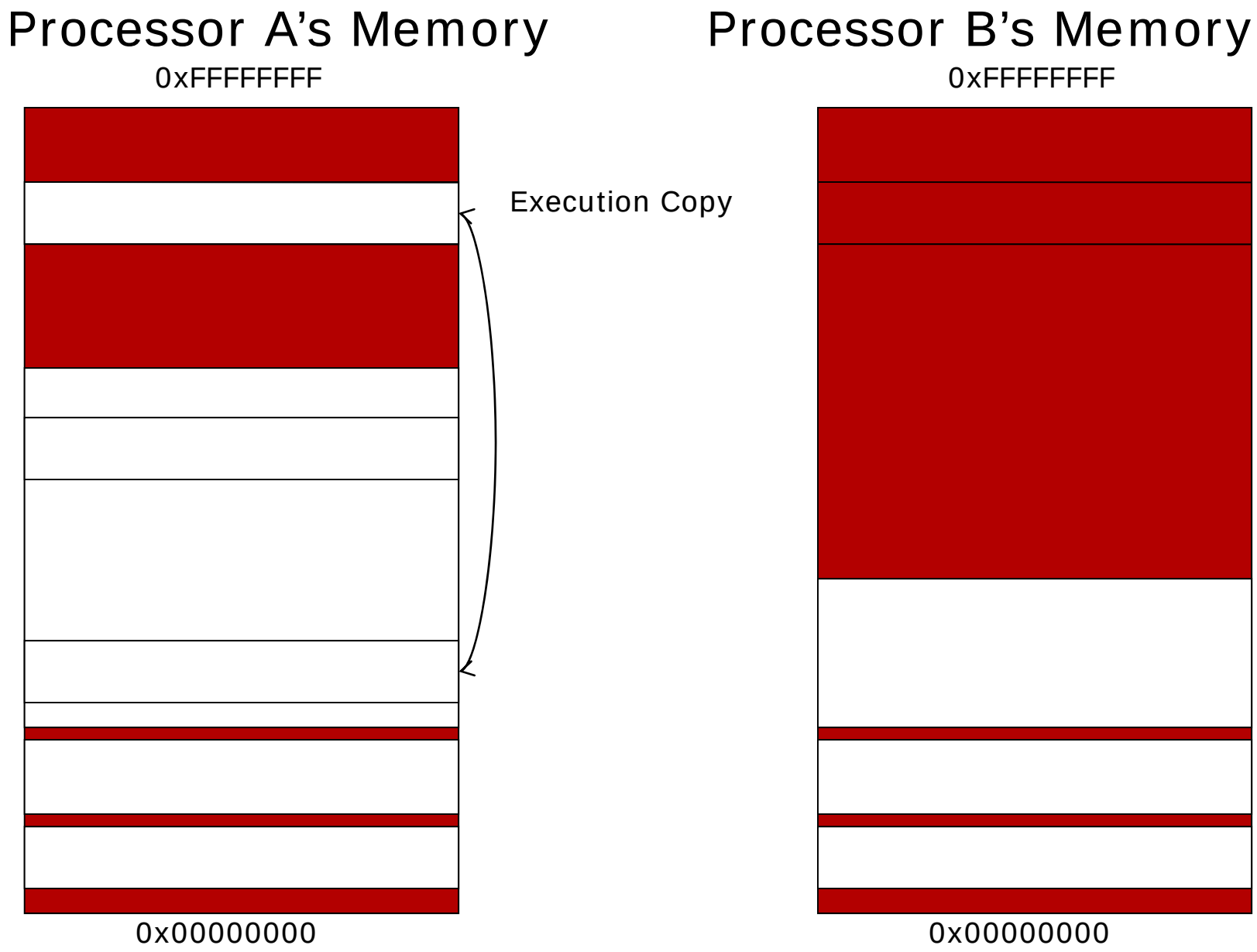
Migrate Stack Data

- **Isomalloc is a completely automatic solution**
  - **No changes needed in application or compilers**
  - **Just like a software shared-memory system, but with proactive paging**
- **But has a few limitations**
  - **Depends on having large quantities of virtual address space (best on 64-bit)**
    - **32-bit machines can only have a few gigs of isomalloc stacks across the whole machine**
  - **Depends on unportable *mmap***
    - **Which addresses are safe? (We must guess!)**
    - **What about Windows? Or Blue Gene?**

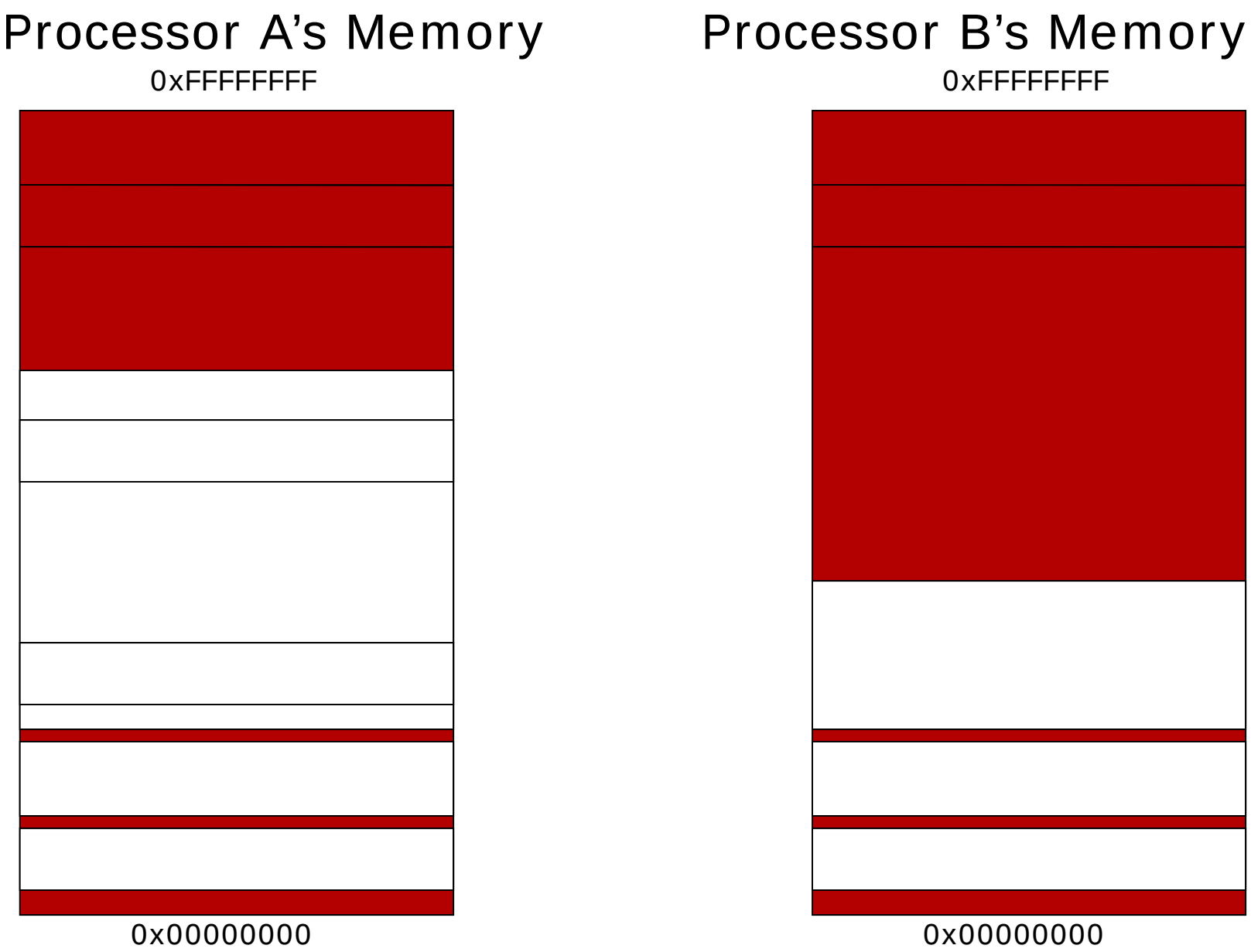
Aliasing Stack Data



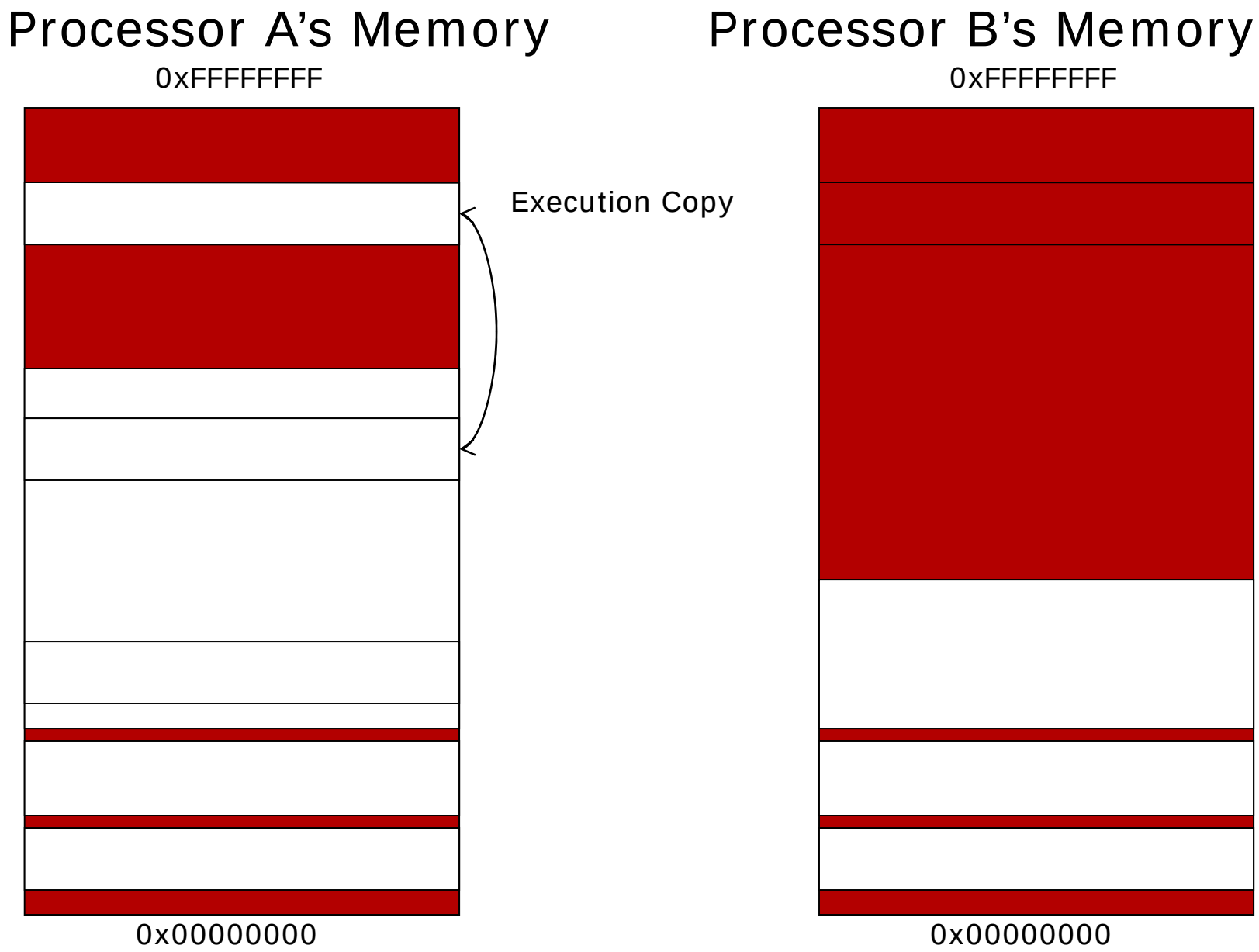
Aliasing Stack Data: Run Thread 2



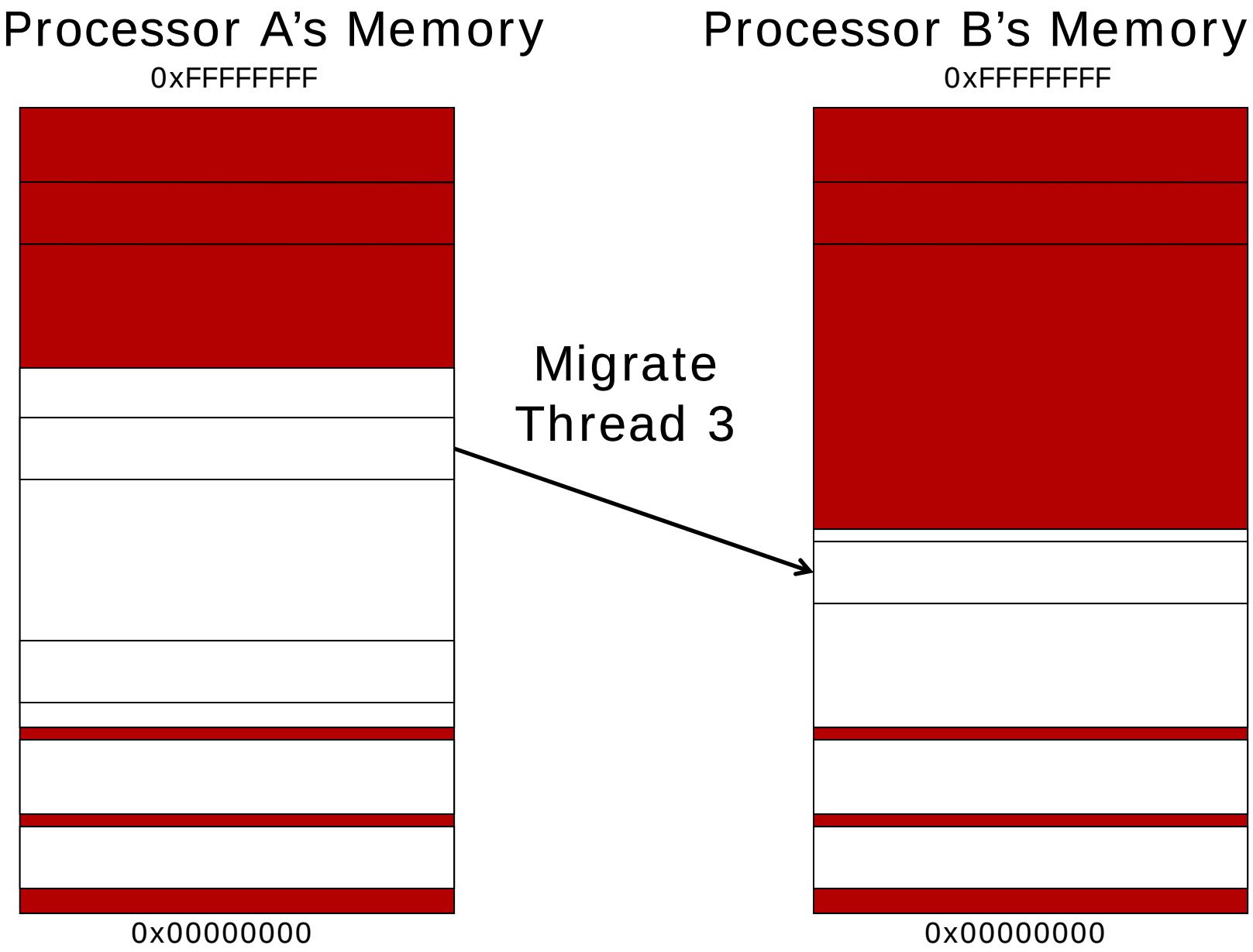
Aliasing Stack Data



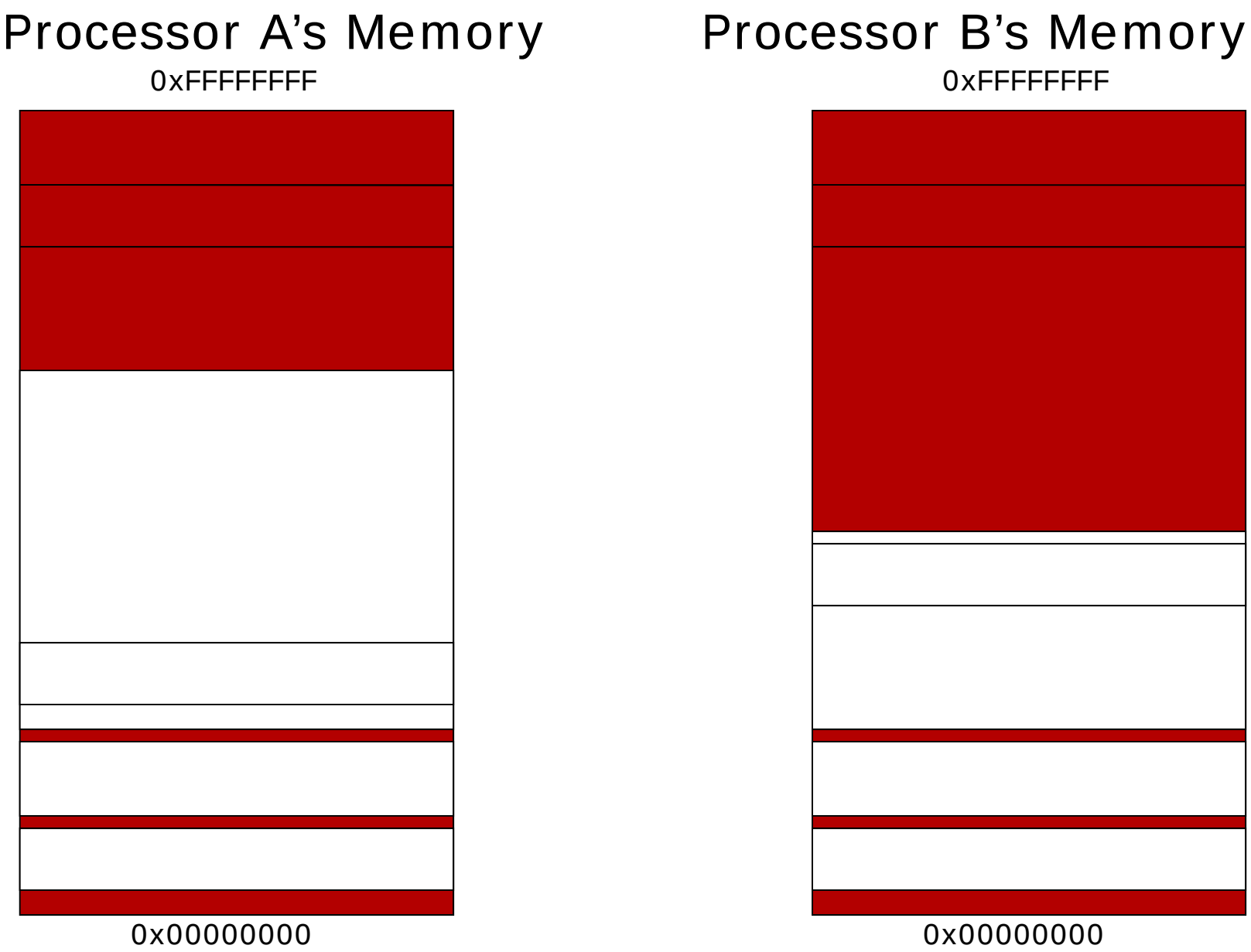
Aliasing Stack Data: Run Thread 3



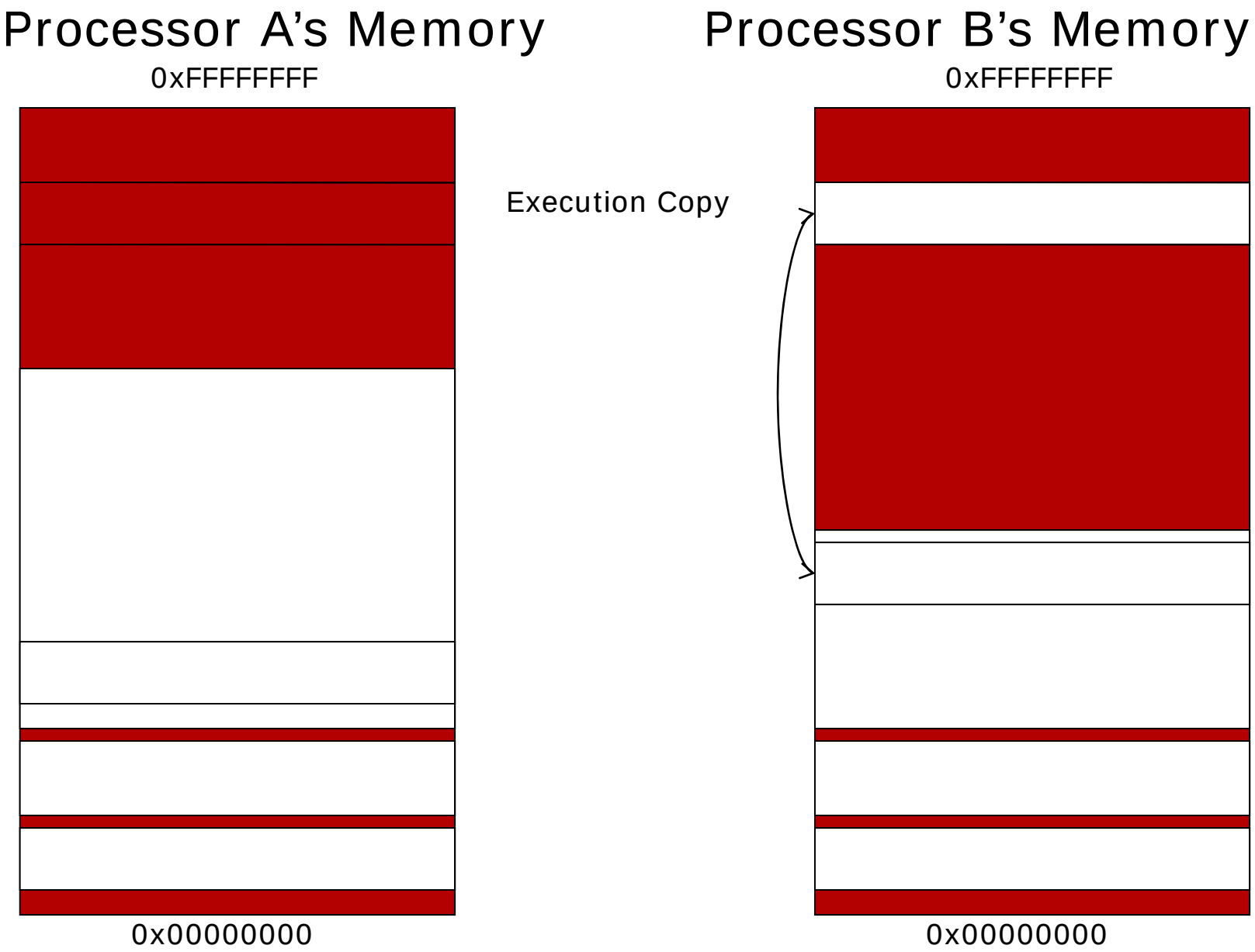
Aliasing Stack Data



Aliasing Stack Data



Aliasing Stack Data



Aliasing Stack Data

- **Does not depend on having large quantities of virtual address space**
  - **Works well on 32-bit machines**
- **Requires only one mmap'd region at a time**
  - **Works even on Blue Gene!**
- **Downsides:**
  - **Thread context switch requires munmap/mmap (3us)**
  - **Can only have one thread running at a time (so no SMP's!)**
- **"-thread memoryalias" link time option**

Heap Data

- **Heap data is any dynamically allocated data**
- **C “malloc” and “free”**
- **C++ “new” and “delete”**
- **F90 “ALLOCATE” and “DEALLOCATE”**
- **Arrays and linked data structures are almost always heap data**

Migrate Heap Data

- **Automatic solution: isomalloc all heap data just like stacks!**
  - **"-memory isomalloc" link option**
  - **Overrides *malloc/free***
  - **No new application code needed**
  - **Same limitations as isomalloc; page allocation granularity (huge!)**
- **Manual solution: application moves its heap data**
  - **Need to be able to size message buffer, pack data into message, and unpack on other side**
  - **"pup" abstraction does all three**

Thank You!

Free source, binaries, manuals, and more information at:

http://charm.cs.uiuc.edu/

Parallel Programming Lab
at University of Illinois