# CS 267 Applications of Parallel Computers

## Lecture 9:

## Split-C

**James Demmel**

http://www.cs.berkeley.edu/~demmel/cs267_Spr99



## Comparison of Programming Models

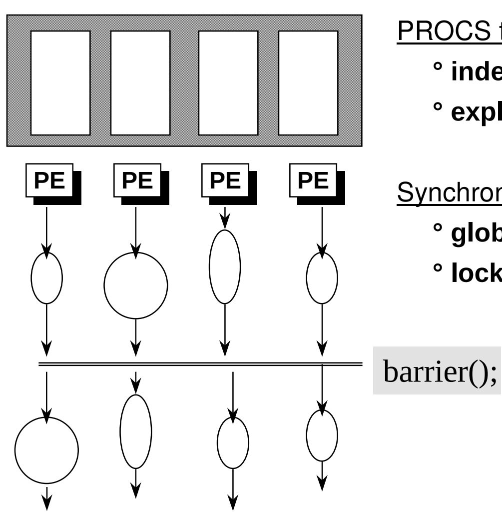
- Data Parallel (HPF)
  - Good for regular applications; compiler controls performance
- Message Passing SPMD (MPI)
  - Standard and portable
  - Needs low level programmer control; no global data structures
- Shared Memory with Dynamic Threads
  - Shared data is easy, but locality cannot be ignored
  - Virtual processor model adds overhead
- Shared Address Space SPMD
  - Single thread per processor
  - Address space is partitioned, but shared
  - Encourages shared data structures matched to the architecture
  - Titanium - targets (adaptive) grid computations
  - Split-C - simple parallel extension to C
- F77 + Heroic Compiler
  - Depends on compiler to discover parallelism
  - Hard to do except for fine grain parallelism, usually in loops

## Overview of Split-C

- Parallel systems programming language based on C
- Can run on most machines
- Creating Parallelism: SPMD
- Memory Model
  - Shared address space via global pointers and spread arrays
- Split phase communication
- Examples and Optimization opportunities

CS267 L9 Split-C Programming.3 Demmel Sp 1999

## SPMD Control Model




PROCS threads of control
- independent
- explicit synchronization

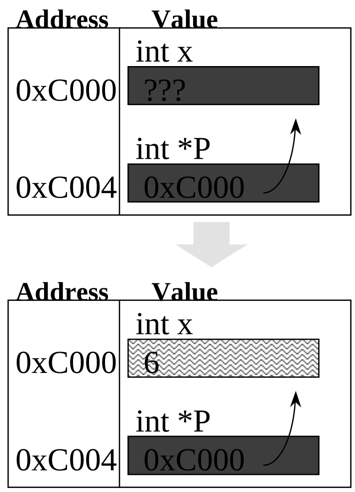
Synchronization
- global barrier
- locks

CS267 L9 Split-C Programming.4 Demmel Sp 1999

## Recall C Pointers

- (&x) read 'pointer to x'
- Types read right to left
  int * read as 'pointer to int'
- *P read as 'value at P'

```
/* assign the value of 6 to x */
int x;
int *P = &x;
*P = 6;
```





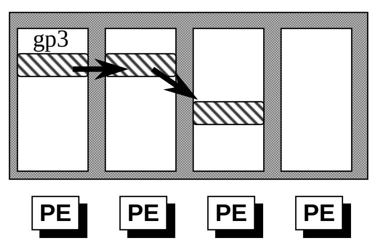

## Global Pointers

A **global** pointer may refer to an object anywhere in the machine.

global ptr = (proc#, local ptr)

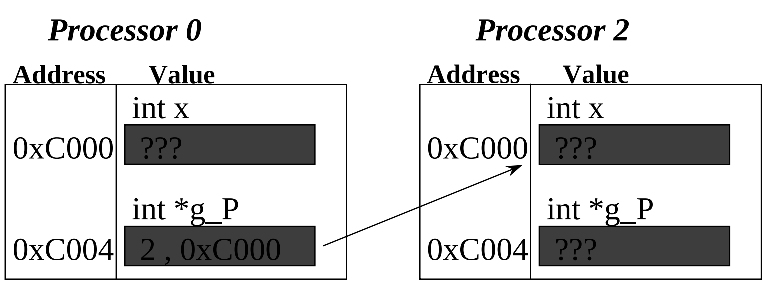
Each object (C structure) pointed to lives on one processor

Global pointers can be dereferenced, incremented, and indexed just like local pointers.

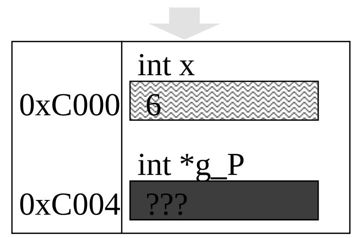



```
int *global gp1;            /* global ptr to an int */
typedef int *global g_ptr;
gptr gp2;                   /* same */
typedef double foo;
foo *global *global gp3;    /* global ptr to a global ptr to a foo*/
int *global *gp4;           /* local ptr to a global ptr to an int*/
```

## Memory Model




```
on_one {
   double *global g_P =
       toglobal(2,&x);
   *g_P = 6;
}
```






## What can global pointers point to?

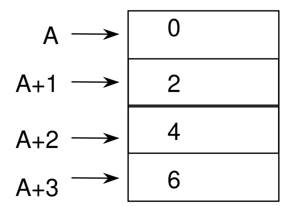
- Anything, but not everything is useful
- Global heap (all_spread_malloc)
  - Space guaranteed to occupy same addresses on all processors so one pointer points to same location relative to base on all processors
- Spread Arrays (coming next)
- Data in stack
  - Dangerous: After routine returns, pointer no longer points to valid address
  - Ok if all processors at same point in call tree

## Recall C Arrays

- Set 4 values to 0,2,4,6
- Origin is 0

```
for (i = 0; i< 4; i++) {
    A[i] = i*2;
}
```

- Pointers & Arrays:
  - A[i] == *(A+i)






## Spread Arrays in Split-C

Spread Arrays are spread over the entire machine

- spreader "::" determines which dimensions are spread
- dimensions to the right define the objects on individual processors
- dimensions to the left are linearized and spread in cyclic map

Example 1: `double A[PROCS]::[10],`

Spread high dimensions    Per processor blocks

A[i][j] is j-th word on processor i

Example 2: `double A[n][r]::[b][b]`
A[i][j] is a b-by-b block living on processor i*r + j mod P

The traditional C duality between arrays and pointers is preserved through *spread* pointers.

## Spread Pointers

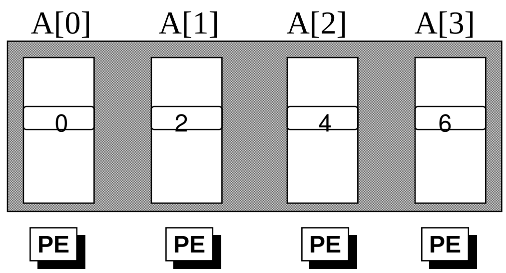
° Global pointers, but with index arithmetic across processors(cyclic)

- If Sptr = (proc,addr) then
  Sptr+1 points to (proc+1, addr)
  (or (0,addr+1) if proc=P-1)

° In contrast if Gptr = (proc,addr) then Gptr+1 points to (proc,addr+1)

No communication:

```
double
   A[PROCS]::;
for_my_1d
   (i,PROCS) {
   A[i] = i*2;}
```






## Blocked Matrix Multiply

```
void all_mat_mult_blk(int n, int r, int m, int b,
                      double C[n][m]::[b][b],
                      double A[n][r]::[b][b],
                      double B[r][m]::[b][b]){
  int i,j,k,l;
  double la[b][b], lb[b][b];
  for_my_2D(i,j,l,n,m) {
    double (*lc)[b] = tolocal(C[i][j]);
    for (k=0;k<r;k++) {
      bulk_read (la, A[i][k], b*b*sizeof(double));
      bulk_read (lb, B[k][j], b*b*sizeof(double));
      matrix_mult(b,b,b,lc,la,lb);
    }
  }
  barrier();
}
```

Configuration independent use of spread arrays

Local copies of subblocks

Highly optimized local routine

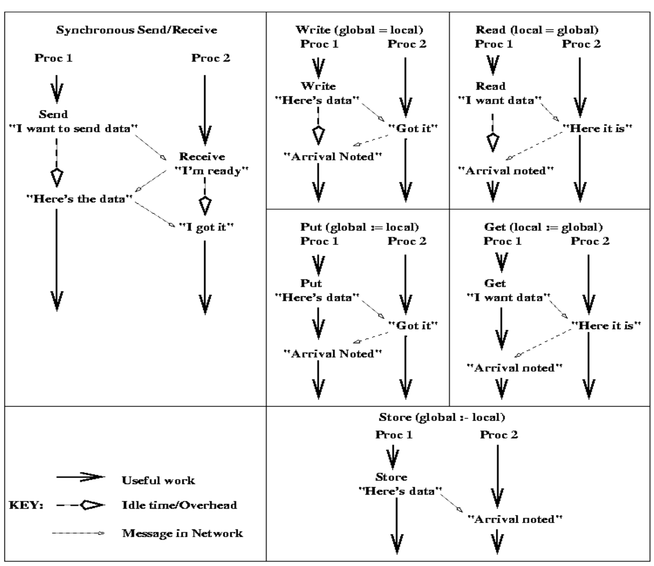
**Blocking improves performance because the number of remote accesses is reduced.**

## Using Global Pointers (Communication)

- Can be used in assignment statements
- Assignment statements can wait to complete, or just initiate communication
- local ptr = global ptr
  - waits to complete read of remote data
- local ptr := global ptr
  - initiates get of remote data
  - can continue computing while communication occurs
  - can do synch later to wait for data to arrive
- global ptr := local ptr
  - similar, but called put



## Message Types in Split-C



## Implementing Communication with Active Messages

° **To implement "local ptr1 := global ptr"**

- **Assume code running on proc 0**
- **Suppose global ptr = (proc#,local ptr2) = (1,local ptr2)**
- **An active message is sent from proc 0 to proc 1, containing**
    - **name of message handler to run on arrival at proc 1**
    - **local ptr 2**
    - **``return address''**
- **When proc 1 receives the active message, it calls the message handler with local ptr2 and return address as arguments**
- **The message handler retrieves the data at local ptr 2 and sends it to the return address**

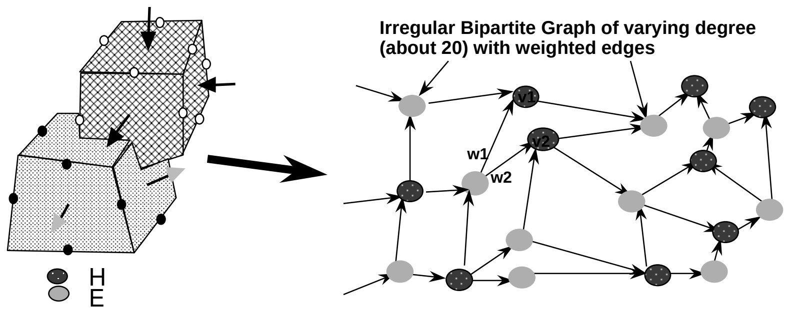
° **The message handler can be very general**

- **Ex: lock(global ptr) returns 1 if the lock is open, 0 otherwise**



## An Irregular Problem: EM3D

**Maxwell's Equations on an Unstructured 3D Mesh**

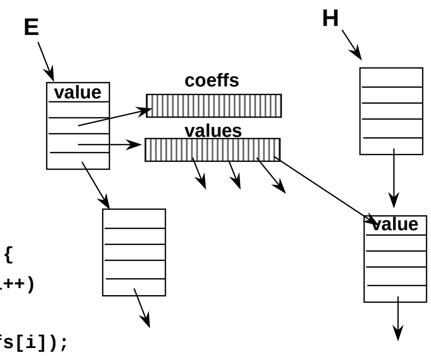



**Basic operation is to subtract weighted sum of neighboring values**

**for all E nodes**
**for all H nodes**

## EM3D: Uniprocessor Version

```
typedef struct node_t {
    double value;
    int edge_count;
    double *coeffs;
    double *(*values);
    struct node_t *next;
} node_t;
void all_compute_E() {
    node_t *n;
    int i;
    for (n = e_nodes; n; n = n->next) {
      for (i = 0; i < n->edge_count; i++)
        n->value = n->value -
          *(n->values[i]) * (n->coeffs[i]);
    }
}
```



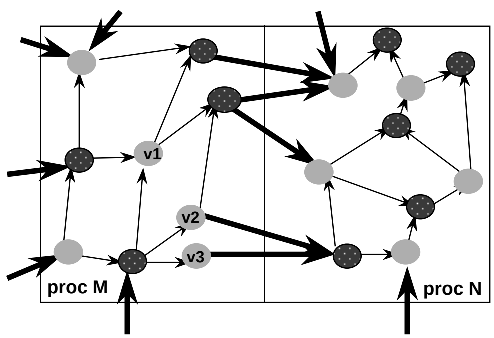

How would you optimize this for a uniprocessor?
- minimize cache misses by organizing list such that neighboring nodes are visited in order



## EM3D: Simple Parallel Version

Each processor has list of local nodes

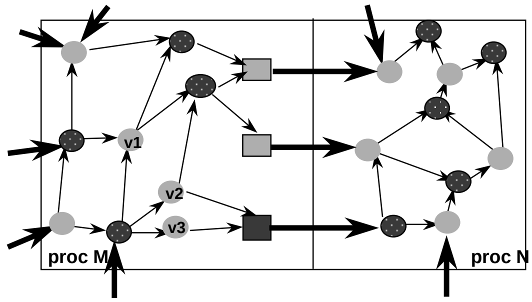
```
typedef struct node_t {
    double value;
    int edge_count;
    double *coeffs;
    double *global (*values);
    struct node_t *next;
} node_t;
void all_compute_e() {
    node_t *n;
    int i;
    for (n = e_nodes; n; n = n->next) {
      for (i = 0; i < n->edge_count; i++)
        n->value = n->value -
          *(n->values[i]) * (n->coeffs[i]);
    }
    barrier();
}
```




How do you optimize this?
- Minimize remote edges
- Balance load across processors:

C(p) = a*Nodes + b*Local Edges + c*Remote Edges

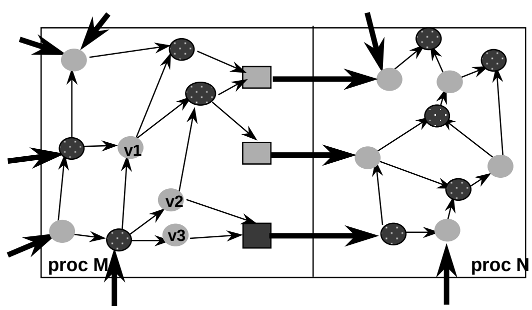
## EM3D: Eliminate Redundant Remote Accesses




```
void all_compute_e()
{   ghost_node_t *g;
    node_t *n;
    int i;
    for (g = h_ghost_nodes; g; g = g->next)  g->value = *(g->rval);
    for (n = e_nodes; n; n = n->next) {
      for (i = 0; i < n->edge_count; i++)
         n->value = n->value - *(n->values[i]) * (n->coeffs[i]);
    }
    barrier();
}
```



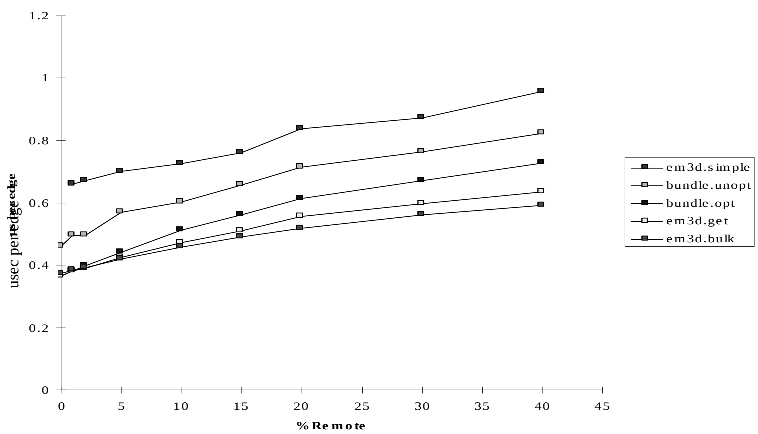
## EM3D: Overlap Global Reads: GET




```
void all_compute_e()
{  ghost_node_t *g;
   node_t *n;
   int i;
   for (g = h_ghost_nodes; g; g = g->next) g->value := *(g->rval);
   sync();
   for (n = e_nodes; n; n = n->next) {
      for (i = 0; i < n->edge_count; i++)
         n->value = n->value - *(n->values[i]) * (n->coeffs[i]);
   }
   barrier();
}
```

## Split-C: Systems Programming

° **Tuning affects application performance**






## Global Operations and Shared Memory

```
int all_bcast(int val) {
   /* broadcast val from processor 0 to all processors */
   int left = 2*MYPROC+1;  /* left child in processor tree */
   int right = 2*MYPROC+2; /* right child in processor tree */
   if (MYPROC > 0) {    /* wait for val from parent */
      while (spread_lock[MYPROC] == 0) {}
      spread_lock[MYPROC] == 0;
      val = spread_buf[MYPROC];
   }
   if ( left < PROCS) {  /* if I have a left child, send val */
      spread_buf[left] = val;
      spread_lock[left] = 1;  /* tell child val has arrived */
   }
   if ( right < PROCS) {  /* if I have a right child, send val */
       spread_buf[right] = val;
       spread_lock[right] = 1;  /* tell child val has arrived */
   }
   return val;
}
```

Requires sequential consistency

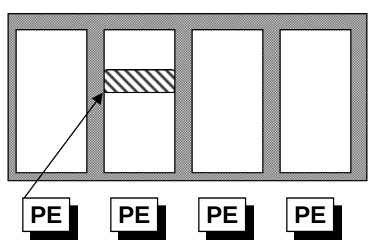
## Global Operations and Signaling Store

```
int all_bcast(int val) {
   /* broadcast val from processor 0 to all processors */
   int left = 2*MYPROC+1;   /* left child in processor tree */
   int right = 2*MYPROC+2;  /* right child in processor tree */
   if (MYPROC > 0) {  /* wait for val from parent */
      store_sync(sizeof(int));
          /* wait until one int has arrived */
      val = spread_buf[MYPROC];
   }
   if ( left < PROCS)  /* if I have a left child, send val */
      spread_buf[left] :- val;
   if ( right < PROCS) /* if I have a right child, send val */
       spread_buf[right] :- val
   return val;
}
```



## Signaling Store and Global Communication

```
void all_transpose ( int m ,
                     double B[PROCS*m],
                     double A[PROCS]::[m])
{
    double *a = &A[MYPROC];
    for (i = 0; i < m; i++) {
       B[m*MYPROC+i] :- a[i];
    }
    all_store_sync();
}
```



## Split-C Library Operations

- Bulk-read, bulk-write, bulk-get, bulk-put, bulk-store
  - More efficient than single word assignment for blocks of data
- Is-sync
  - Returns a boolean indicated whether all outstanding get/puts are done
- All_spread_free
  - To go with all_spread_malloc
- All_reduce_to_one
  - Reduction operations (add, max, etc.)
- All_scan
  - Parallel prefix
- Fetch_and_add, exchange, test_and_set, cmp_and_swap
  - Atomic operations for synchronization



## Split-C Summary

- Performance tuning capabilities of message passing
- Support for shared data structures
- Installed on NOW and available on most platforms
  - http://www.cs.berkeley.edu/projects/parallel/castle/split-c
- Consistent with C design
  - arrays are simply blocks of memory
  - no linguistic support for data abstraction
    - interfaces difficult for complex data structures
  - explicit memory management