
CS 267 Applications of Parallel Computers

Lecture 7: Programming with Threads and Comparison on Fish

2/12/97

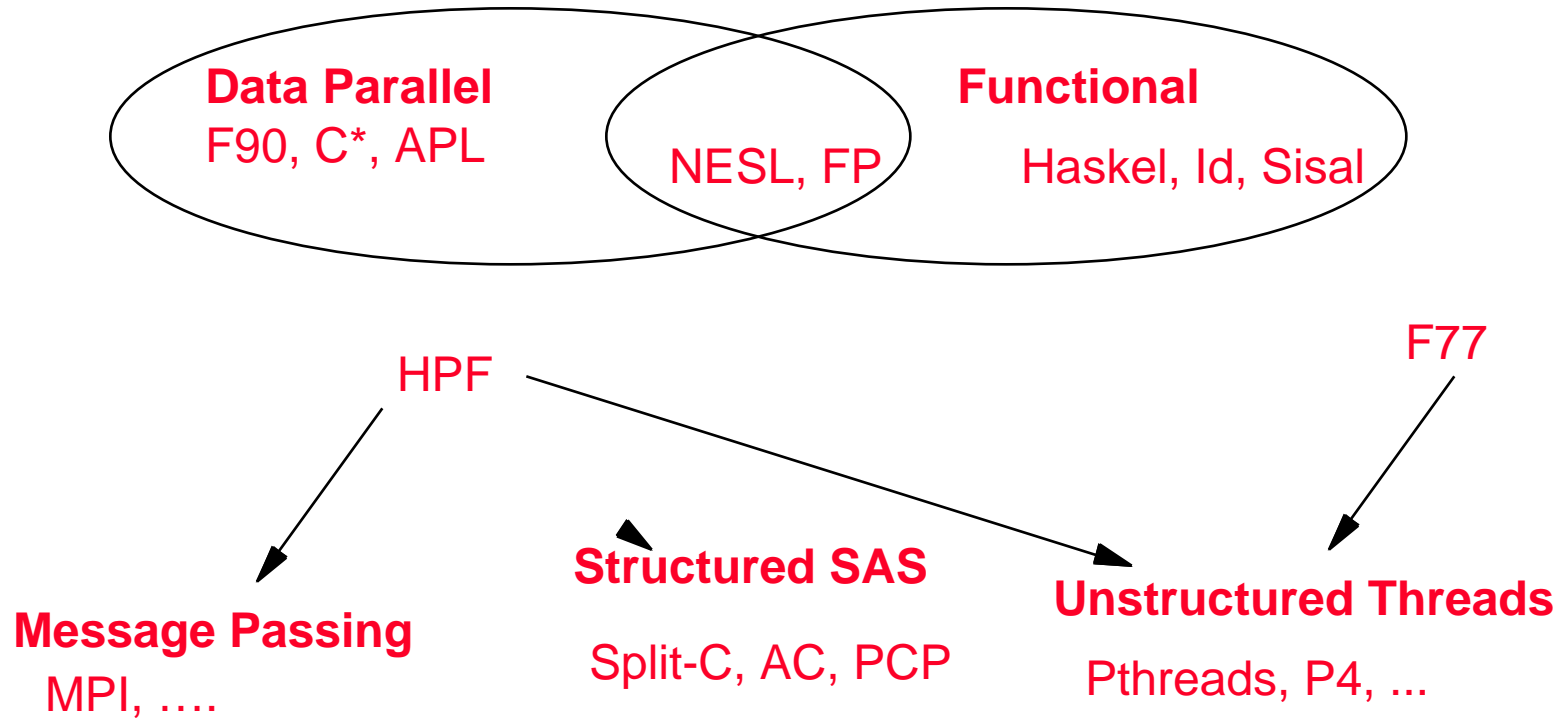
David E. Culler

<http://www.cs.berkeley.edu/cs267/>

Outline

- **Recap**
- **Perspective on Programming Languages**
- **Programming with Threads**
- **Discussion of Performance on Matrix Multiply**
- **Comparisons**

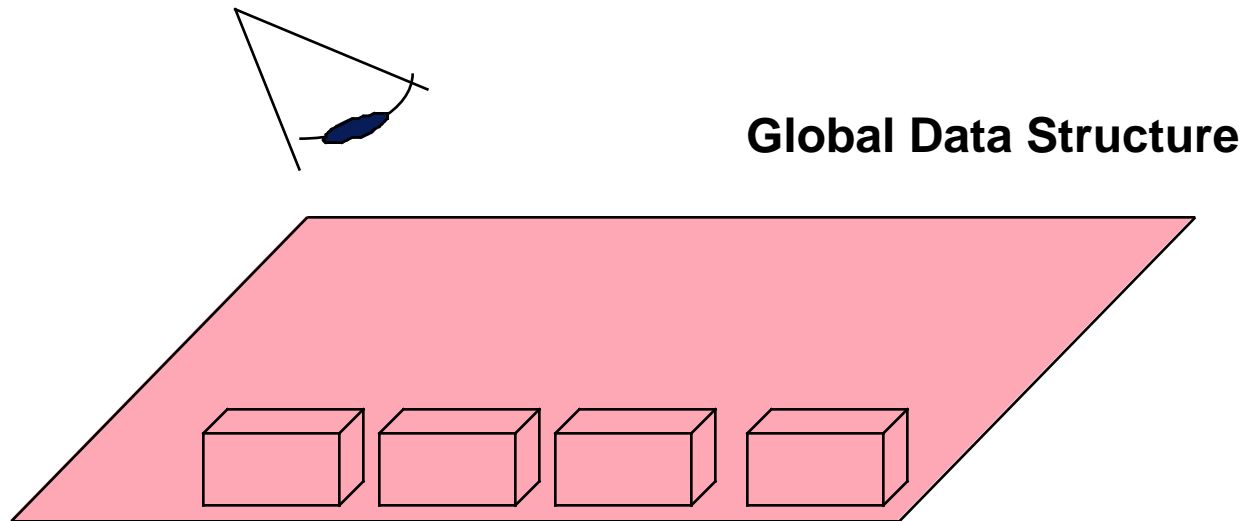
Parallel Programming Languages



Recall: Basic Questions

- **How is Parallelism Expressed?**
- **How is Communication Expressed?**
- **How is Synchronization Expressed?**
- **What global data structures can be created?**
- **How do you optimize for Performance?**

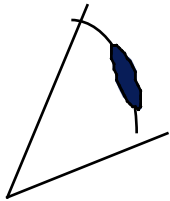
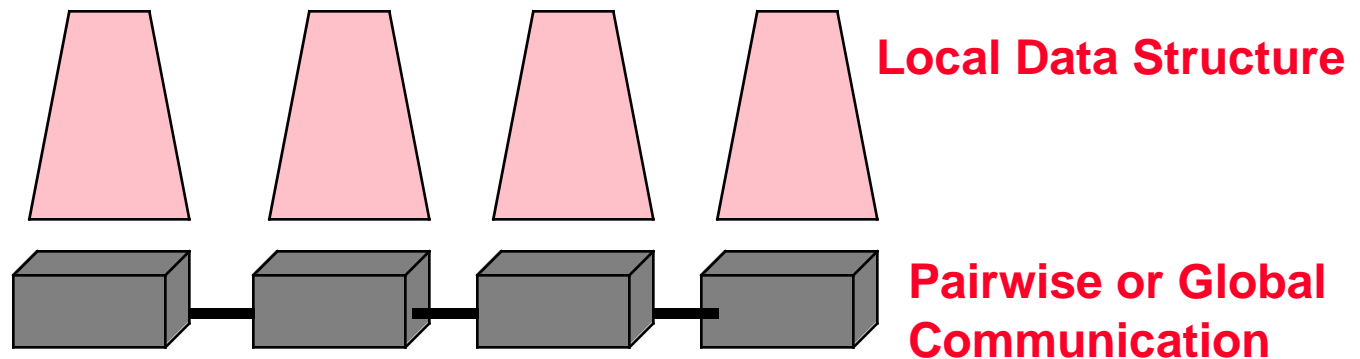
Data Parallel View



- **Programmers view is of a sequence of large scale transformations to global data structures**
- **Processors/Threads are hidden from view**
- **HPF “directives” lower the level**

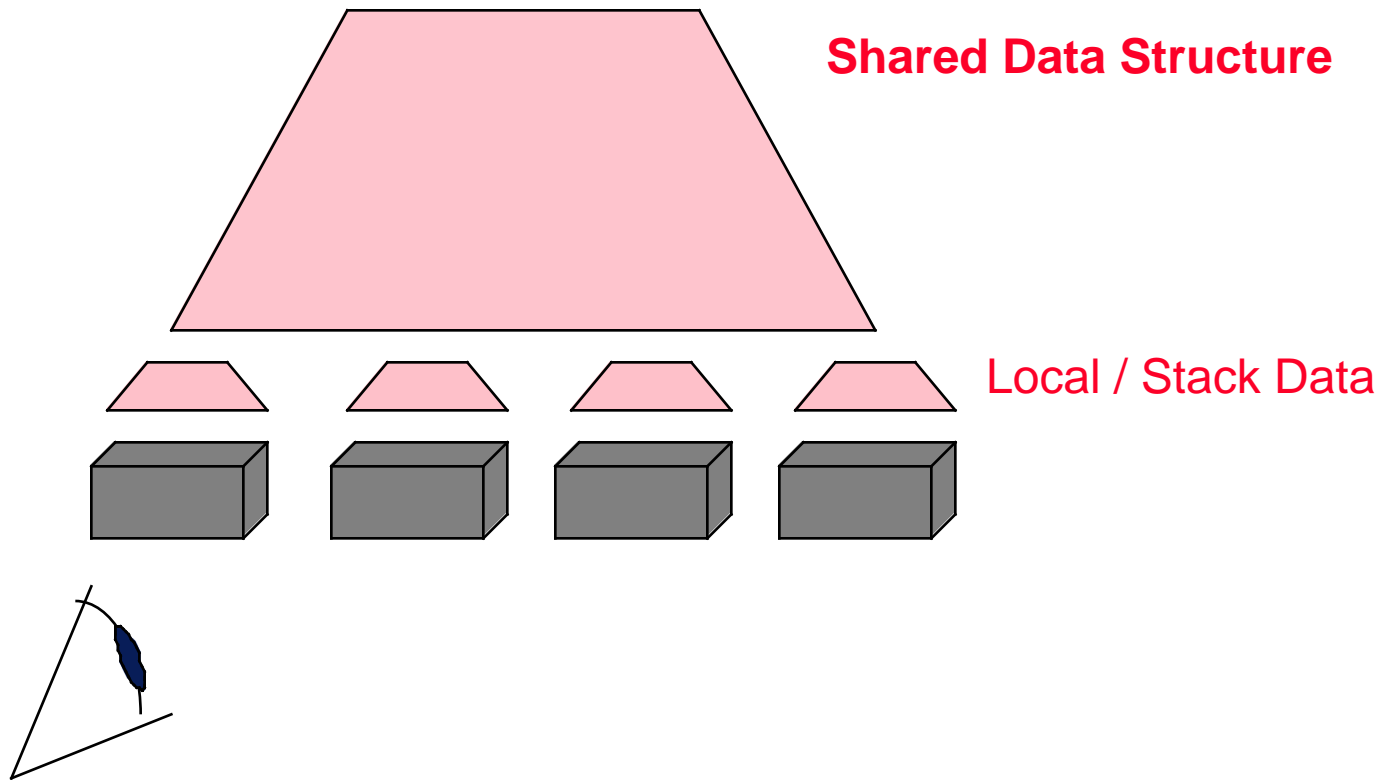
Message Passing View

- Programmers view is entirely processor-centric
- Specifies what each processor/thread does
- Global view implicit in local data + communication pattern



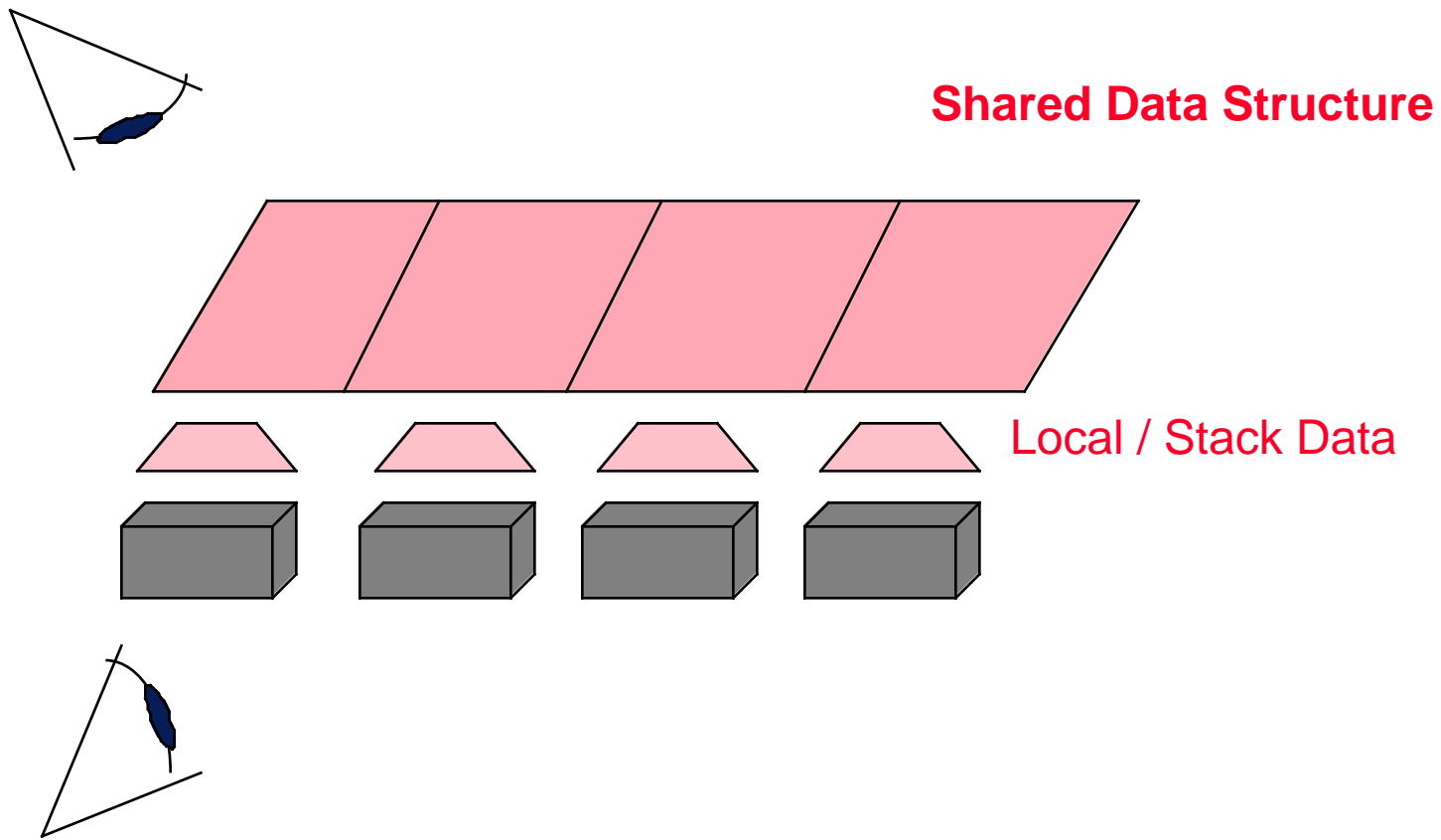
Uniform Shared Address Space

- Programmers view is still processor-centric
- Specifies what each processor/thread does
- Global view implicit in pattern of data sharing



Segmented Shared Address Space

- Programmer has local and global view
- Specifies what each processor/thread does
- Global data, operation, and synchronization



Programming with Threads

- **Several Threads Libraries**
 - much like msg passing situation before MPI
- **PTHREADS is the Posix Standard**
 - Solaris threads are very similar
- **P4 (Parmacs) is a widely used portable package**
- **Few basic components needed for (structured) parallel programming**
 - huge number of details for systems use
- **Thread creation / termination**
- **Shared data**
- **Synchronization operations**

Thread Creation

- **F90: single thread starts at main routine and goes to the end**
- **MPI and Split-C: all threads start at main routine and go to the end**
 - serialize by some of them waiting
- **Solaris / P4**
 - single master thread starts and spawns other “workers”

Example: Thread Creation in P4

```
#include "p4.h"
```

```
main (int argc, char **argv)
```

```
{
```

```
    p4_initenv (&argc, argv);
```

```
    if (p4_get_my_id() == 0) p4_create_procgrouop(); /* create rest */
```

```
    worker();
```

```
    p4_wait_for_end();
```

```
}
```

```
worker ();
```

```
{
```

```
    printf("Hello from %d \n", p4_get_my_id());
```

```
}
```

With Solaris Threads

```
main()
{
    thread_ptr = (thrinfo_t *) malloc(NTHREADS * sizeof(thrinfo_t));
    thread_ptr[0].chunk = 0;
    thread_ptr[0].tid = myID;
    for (i = 1; i < NTHREADS; i++) {
        thread_ptr[i].chunk = i;
        if (thr_create(0, 0, worker, (void*)&thread_ptr[i].chunk,
                    0, &thread_ptr[i].tid)) {
            perror("thr_create");
            exit(1);
        }
    }
    worker(0);
    for (i = 1; i < NTHREADS; ++i)
        thr_join(thread_ptr[i].tid, NULL, NULL);
}
```

Discussion

- **There are two basic approaches to creating threads**
 - **fork a thread within the same process sharing the same address space (PTHREADS)**
 - **spawn a process (with the same image) and rendezvous to obtain a shared region of the address space (SYS V)**

Shared Address Allocation

- **Most systems provide a special form of malloc/free**
 - p4_shmalloc, p4_shfree
 - p4_malloc, p4_free are just basic malloc/free
 - sharing unspecified
- **Solaris threads**
 - malloc'd and static variables are shared

Synchronization

- **Can build it yourself out of flags**
 - `while (!flag) {};`
- **Lock/Unlock primitives build in the waiting**
 - typically well tested and optimized for the machine
 - system friendly
- **Most systems provide higher level synchronization primitives**
 - barrier - global synchronization
 - semaphores
 - monitors

Solaris Threads Example

```
mutex_t  mul_lock;
barrier_t ba;
int sum;
main()
{
    sync_type = USYNC_PROCESS;
    mutex_init(&mul_lock, sync_type, NULL);
    barrier_init(&ba, NTHREADS, sync_type, NULL
.... spawn all the threads as above...
}
worker (int me)
{
    int x = all_do_work(me);
    barrier_wait(&ba);
    mutex_lock(&mul_lock);
    sum += mine;
    mutex_unlock(&mul_lock);
}
```

Data Decomposition

- **F90: Unspecified**
 - left to the compiler to determine
- **HPF: Programmer directed layout hints**
 - `real D (1024), E(1024)`
 - `!HPF$ DISTRIBUTE D (BLOCK)`
 - `!HPF$ DISTRIBUTE E (CYCLIC)`
- **MPI: Inherent in local data**
 - `double D[n/PROCS], E[n/PROCS]`
- **Split-C: Determined by canonical data layout**
 - `double D[PROCS]::[n/PROCS], E[1024]::;`
- **P4/Pthreads: Unspecified**
 - assumed globally accessible

Break for Matrix Multiply Performance

How did you do and what did you have to do to get it?

- **Blocking**
- **Pay attention of data placement**
 - reuse data while in cache
 - registers
- **Utilize large transfers**
 - use full cache line while you've got it
- **Exploit instruction level parallelism**
- **Avoid pipeline delays**

Let's look at Shark&Fish 1

- **Fish swim independently under the force of a current**
- **Time step such that no one swims to far**
 - one cell

F90 and HPF

- <http://now.CS.Berkeley.EDU/cs267/assignment3/hpf/fish1/fish1.hpf>

Split-C

MPI

Solaris Threads

Work Distribution

- **The Split-C, MPI, and Threads version all use static assignment of the iteration space**
 - without global data structures, must convert index
- **The HPF versions “suggests” the same to the compiler**
- **Code must contain work assignment whether or not describes data assignment**

```
for (I = MyMin; I<=MyMax; I++) {  
    A[I] = f(I);  
}
```

```
for (I = 0; I < MyCount; I++) {  
    A[I] = f(I+MyMin);  
}
```

What about cyclic assignment

```
for (I = MyProc; I < n; I += PROCS) {  
    A[I] = f(I);  
}
```

```
for (I = 0; I < MyCount; I++) {  
    A[I] = f( ??? );  
}
```

```
for_my_1D(I,n) {  
    A[I] = f(I);  
}
```

- **Assignment of work is easier in a global address space**
- **It is faster if it corresponds to the data placement!**
- **Hardware replication moves data to where it is accessed**

Self Scheduling

```
while (fetch&add (I) < n) {  
    A[I] = f(I);  
}
```

- Impact on load balancing?
- Impact on data access?

How do you get performance in each model?

- more on friday