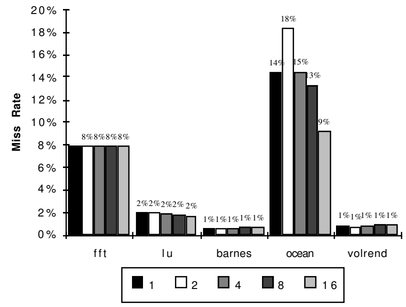
Miss Rates for Snooping Protocol

- 4th C: Coherency Misses
- More processors: increase Coherency, decrease Capacity



Fast Fourier Transform (FFT): Matrix transposes + comp.

LU factorization of dense 2D matrix (linear algebra)

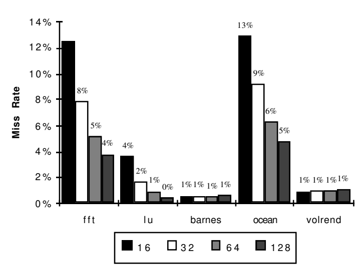
Barnes-Hut n-body algorithm solving galaxy evolution probem

Ocean simluates influence of eddy & boundary currents on large-scale flow in ocean: dynamic arrays per grid

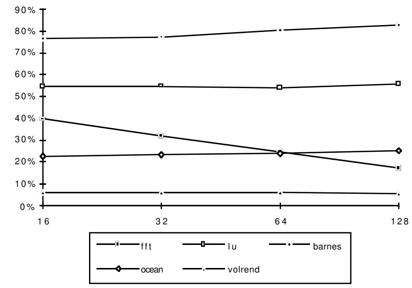
VolRend is parallel volume rendering: scientific visualization

Block Size

- Since cache block hold multiple words, may get coherency traffic for unrelated variables in same block
- False sharing arises from the use of an invalidation-based coherency algorithm. False sharing occurs when a block is invalidated (and a subsequent reference causes a miss) because some word in the block, other than the one being read, is written into.

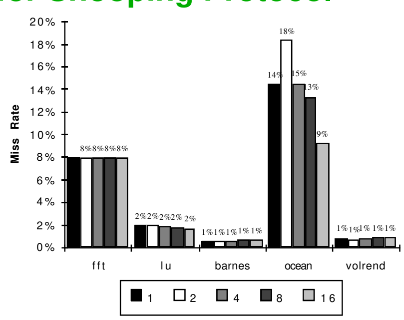


% Misses caused by cohernecy traffic vs. Block size



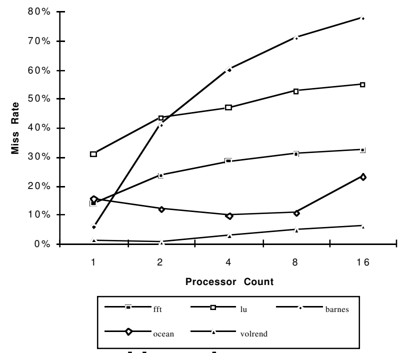
- **FFT communicates data in large blocks & communication adapts to the block size (it is a parameter to the code); makes effective use of large blocks.**
- **Ocean competing effects that favor different block size**
  - **accesses to the boundary of each subgrid, in one direction the accesses match the array layout, taking advantage of large blocks, while in the other dimension, they do not match. These two effects largely cancel each other out leading to an overall decrease in the coherency misses as well as the capacity misses.**

Miss Rates for Snooping Protocol



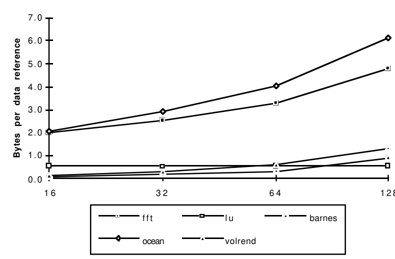
- **Cache size is 64KB, 2-way set associative, with 32B blocks.**
- **With the exception of Volrend, the misses in these applications are generated by accesses to data that is potentially shared.**
- **Except for Ocean, data is heavily shared; in Ocean only the boundaries of the subgrids are shared, though the entire grid is treated as a shared data object. Since the boundaries change as we increase the processor count (for a fixed size problem), different amounts of the grid become shared. The analmous increase in miss rate for Ocean in moving from 1 to 2 processors arises because of conflict misses in accessing the subgrids.**

% Misses caused by cohernecy traffic vs. Processors



- % cache misses caused by coherency transactions typically rises when a fixed size problem is run on more processors.
- The absolute number of coherency misses is increasing in all these benchmarks, including Ocean. In Ocean, however, it is difficult to separate out these misses from others, since the amount of sharing of the grid varies with processor count.
- Invalidations increases significantly; FFT the miss rate arising from coherency misses increases from nothing to almost 7%.

Bus Traffic as increase Block Size



- Bus traffic climbs steadily as the block size is increased.
- Volrend the increase is more than a factor of 10, although the low miss rate keeps the absolute traffic small.
- The factor of 3 increase in traffic for Ocean is the best argument against larger block sizes.
- Remember that our protocol treats ownership misses the same as other misses, slightly increasing the penalty for large cache blocks: in both Ocean and FFT this effect accounts for less than 10% of the traffic.

Synchronization

- Why Synchronize? know when its safe for different processes to use shared data
- 3 Issues for Syncronization

1) Uninterruptable instruction to fetch and update memory (atomic operation)

2) User level synchronization operation using this primitive

3) For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

1) Uninterruptable instruction to fetch and update memory

- **Atomic exchange**: interchange a value in a register for a value in memory
  - 0 => synchronization variable is free
  - 1 => synchronization variable is locked and unavailable
  - Set register to 1 & swap
  - New value in register determines success in getting lock
    0 if you are first
    1 if some other processor had already claimed access
  - key is that exchange is indivisible
- **Test-and-set**: tests a value and sets it if the value passes the test
- **Fetch-and-increment**: it returns the value of a memory location and atomically increments it
  - 0 => synchronization variable is free

1) Uninterruptable instruction to fetch and update memory

- Hard to have tread & write in one instructio: use 2 instead
- load linked (or load locked) + store conditional
  - load linked returns the initial value
  - store conditional returns 1 if it succeeds and 0 otherwise
- Example doing atomic swap with LL & SC:

```
try:    mov     R3,R4           ;mov exchange value
        ll      R2,0(R1)        ;load linked
        sc      R3,0(R1)        ;store
        beqz    R3,try          ;branch store fails
        mov     R4,R2           ;put load value in R4
```

- Example doing fetch & increment with LL & SC:

```
try:    ll      R2,0(R1)        ;load linked
        addi    R2,R2,#1        ;increment (OK if reg–reg)
        mov     R3,R2           ;save value (OK if reg–reg)
        sc      R2,0(R1)        ;store
        beqz    R2,try          ;branch store fails
```


2) User level synchronization operation using this primitive

- spin locks: processor continuously tries to acquire, spinning around a loop trying to get the lock

```
            li      R2,#1
lockit:     exch    R2,0(R1)      ;atomic exchange
            bnez    R2,lockit     ;already locked?
```

- What about MP with cache coherency?
  - Want to spin on cache copy
  - Likely to get cache hits for such variables
- Problem: exchange includes a write, which invalidates all other copies => lots of bus traffic
- Solution: first repeated just read variable; when it changes, then try exchange:

```
try:        li      R2,#1
lockit:     lw      R3,0(R1)      ;load var
            bnez    R3,lockit     ;not free=>spin
            exch    R2,0(R1)      ;atomic exchange
            bnez    R2,try        ;already locked?
```

Steps for Invalidate Protocol

Step	*P0*	*$*	*P1*	*$*	*P2*	*$*	*Bus/Direct activity*
1.	Has lock	Sh	spins	Sh	spins	Sh	None
2.	Lock<– 0	Ex		Inv		Inv	P0 Invalidates lock
3.		Sh	miss	Sh	miss	Sh	WB P0; P2 gets bus
4.		Sh	waits	Sh	lock = 0	Sh	P2 cache filled
5.		Sh	lock=0	Sh	exch	Sh	P2 cache miss(WI)
6.		Inv	exch	Inv	r=0;l=1	Ex	P2 cache filled; Inv
7.		Inv	r=1;l=1	Ex	locked	Inv	WB P2; P1 cache
8.		Inv	spins	Ex		Ex	None

3) For large scale MPs, synchronization can be a bottleneck

- Example: 20 processors try to lock, 50 cycles for bus
  Read miss all waiting processors to fetch lock 1000
  Write miss by releasing processor and invalidates 50
  Read miss by all waiting processors 1000
  Write miss by all waiting processors ,
  one successful lock, & invalidate all copies 1000
  Total time for 1 proc. to acquire & release lock 3050
- Each time one gets a lock, drops out of competition= 1525
- 20 x 1525 = 30,000 cycles for 20 processors to pass through the lock
- Problem is contention for lock and serialization of lock access: once lock is fre all compete to see who gets it
- Alternative: create a list of waiting processors, go through list: called a "queuing lock"
  – special HW to recognize 1st lock access & lock release
- Another laternative: fetch-and-increment; can be used to create barrier; wait until everyone reaches same point

Another MP Issue: Memory Consistency Models

- **How consistent? When must a processor see the new value? e.g., seems that**

P1:	**A = 0;**		**P2:**	**B = 0;**
	.....			**.....**
	A = 1;			**B = 1;**
L1:	**if (B == 0) ...**		**L2:**	**if (A == 0) ...**

- **Impossible for both if statements L1 & L2 to be true?**
  - **What if write invalidate is delayed & processor continues?**
- **Memory consitency models: what are the rules for such cases**
- **Sequential consistency: result of any execution be the same as if the accesses of each processor were kept in order and the accesses among different processors were interleaved => assignments before ifs above**
- **SC => delay all memory accesses until all invalidates done**

Memory Consistency Model

- **Schemes faster execution to sequential consistency**
- **Not really an issue for most programs; they are synchronized**
  - **A program is synchronized if all access to shared data are ordered by synchronization operations**

    write (x)
    ...
    release (s) ***{unlock}***
    ...
    acquire (s) ***{lock}***
    ...
    read(x)
- **Only those programs willing to be nondeterminisitc are not sychronized**
- **Several Relaxed Models for Memory Consistency since most programs are synchronized: charcerized by their attitude towards: RAR, WAR, RAW, WAW to different addresses**

Key Issues for MPs

- **Measuring Performance**
  - not just time on one size, but how performance scales with P
  - for fixed size problem (same memory per processor) and scaled up problem (fixed execution time)
  - care to compare to best uniprocessor algorithm, not just parallel program on 1 processor (unless its best)
- **Multilevel caches, cohernecy, and inclusion**
  - get duplication by using coerhency at level of second cache provided data of all higher levels are included
  - block size of L1 & L2?
- **Nonblock caches and Prefetching**
  - more latency to hide so nonblocking more important
  - predicated on free memory bandwidth, may be in conflict with other processors
  - want prefetch to be coherent (“nonbinding” to local copy)
- **Virtual Memory to get Shared Memory MP: Distributed VIrtual Memory (DVM); pages are units of coherency**