Hardware/Software Performance Tradeoffs
(plus Msg Passing Finish)

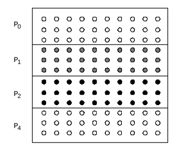
CS 258, Spring 99

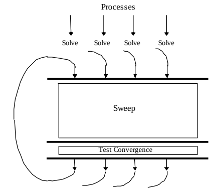
David E. Culler

Computer Science Division

U.C. Berkeley

SAS Recap



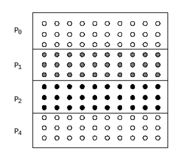


- **Partitioning = Decomposition + Assignment**
- **Orchestration = coordination and communication**
  - **SPMD, Static Assignment**
  - **Implicit communication**
  - **Explicit Synchronization: barriers, mutex, events**

Message Passing Grid Solver

- **Cannot declare A to be global shared array**
  - **compose it logically from per-process private arrays**
  - **usually allocated in accordance with the assignment of work**
    - » **process assigned a set of rows allocates them locally**
- **Transfers of entire rows between traversals**
- **Structurally similar to SPMD SAS**
- **Orchestration different**
  - **data structures and data access/naming**
  - **communication**
  - **synchronization**
- **Ghost rows**

Data Layout and Orchestration



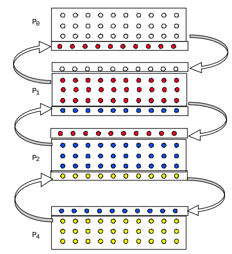
Data partition allocated per processor

Add ghost rows to hold boundary data

Send edges to neighbors

Receive into ghost rows

Compute as in sequential program




```
10.   procedure Solve()
11.   begin
13.      int i,j, pid, n' = n/nprocs, done = 0;
14.      float temp, tempdiff, mydiff = 0;          /*private variables*/
6.    myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
                                                    /*initialize my rows of A, in an unspecified way*/

15.   while (!done) do
16.      mydiff = 0;                                /*set local diff to 0*/
      /* Exchange border rows of neighbors into myA[0,*] and myA[n'+1,*]*/
16a.     if (pid != 0) then SEND(&myA[1,0], n*sizeof(float),pid-1, ROW);
16b.     if (pid = nprocs-1) then
            SEND(&myA[n', 0], n*sizeof(float),pid+1, ROW);
16c.     if (pid != 0) then RECEIVE(&myA[0,0], n*sizeof(float),pid-1, ROW);
16d.     if (pid != nprocs-1) then
            RECEIVE(&myA[n'+1,0], n*sizeof(float), pid+1, ROW);
17.      for i ← 1 to n' do                         /*for each of my (nonghost) rows*/
18.         for j ← 1 to n do                       /*for all nonborder elements in that row*/
19.            temp = myA[i,j];
20.            myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.               myA[i,j+1] + myA[i+1,j]);
22.            mydiff += abs(myA[i,j] - temp);
23.         endfor
24.      endfor
                                                    /*communicate local diff values and determine if
                                                    done; can be replaced by reduction and broadcast*/
25a.        if (pid != 0) then                      /*process 0 holds global total diff*/
25b.           SEND(mydiff,sizeof(float),0, DIFF);
25c.           RECEIVE(done,sizeof(int),0, DONE);
25d.        else                                    /*pid 0 does this*/
25e.           for i ← 1 to nprocs-1 do             /*for each other process*/
25f.              RECEIVE(tempdiff,sizeof(float),*, DIFF);
25g.           mydiff += tempdiff;                  /*accumulate into total*/
25h.           endfor
25i            if (mydiff/(n*n) < TOL) then done = 1;
25j.           for i ← 1 to nprocs-1 do             /*for each other process*/
25k.              SEND(done,sizeof(int), i, DONE);
25l.           endfor
25m.     endif
26.   endwhile
27.   end procedure
```


Notes on Message Passing Program

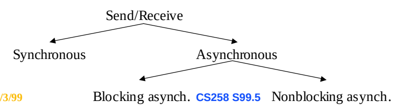
- **Use of ghost rows**
- **Receive does not transfer data, send does**
  - **unlike SAS which is usually receiver-initiated (load fetches data)**
- **Communication done at beginning of iteration, so no asynchrony**
- **Communication in whole rows, not element at a time**
- **Core similar, but indices/bounds in local rather than global space**
- **Synchronization through sends and receives**
  - **Update of global diff and event synch for done condition**
  - **Could implement locks and barriers with messages**
- **REDUCE and BROADCAST simplify code**

```
           /*communicate local diff values and determine if done, using reduction and broadcast*/
25b.      REDUCE(0,mydiff,sizeof(float),ADD);
25c.      if (pid == 0) then
25i.         if (mydiff/(n*n) < TOL) then done = 1;
25k.      endif
25m          BROADCAST(0,done,sizeof(int),DONE);
```


NOW Handout Page 1

Send and Receive Alternatives

- extended functionality: stride, scatter-gather, groups
- Sychronization semantics
  - Affect when data structures or buffers can be reused at either end
  - Affect event synch (mutual excl. by fiat: only one process touches data)
  - Affect ease of programming and performance
- Synchronous messages provide built-in synch. through match
  - Separate event synchronization may be needed with asynch. messages
- With synch. messages, our code may hang. Fix?



Orchestration: Summary

- Shared address space
  - Shared and private data (explicitly separate ??)
  - Communication implicit in access patterns
  - Data distribution not a correctness issue
  - Synchronization via atomic operations on shared data
  - Synchronization explicit and distinct from data communication
- Message passing
  - Data distribution among local address spaces needed
  - No explicit shared structures
    - » implicit in comm. patterns
  - Communication is explicit
  - Synchronization implicit in communication
    - » mutual exclusion by fiat

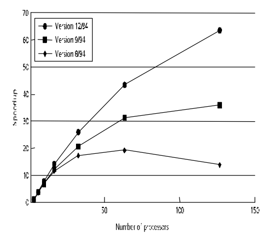
Correctness in Grid Solver Program

	SAS	Msg-Passing
Explicit global data structure?	Yes	No
Assignment indept of data layout?	Yes	No
Communication	Implicit	Explicit
Synchronization	Explicit	Implicit
Explicit replication of border rows?	No	Yes

- Decomposition and Assignment similar in SAS and message-passing
- Orchestration is different
  - Data structures, data access/naming, communication, synchronization
  - Performance?

Performance Goal => Speedup

- Architect Goal
  - observe how program uses machine and improve the design to enhance performance
- Programmer Goal
  - observe how the program uses the machine and improve the implementation to enhance performance
- What do you observe?
- Who fixes what?



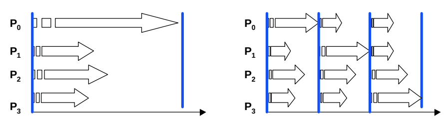
Analysis Framework

$$\text{Speedup} \leq \frac{\text{Sequential Work}}{Max\ (\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})}$$

- Solving communication and load balance NP-hard in general case
  - But simple heuristic solutions work well in practice
- Fundamental Tension among:
  - balanced load
  - minimal synchronization
  - minimal communication
  - minimal extra work
- Good machine design mitigates the trade-offs

Load Balance and Synchronization

$$\text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{Max\ \text{Work on any Processor}}$$

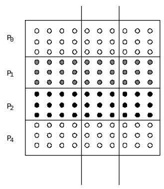


- Instantaneous load imbalance revealed as wait time
  - at completion
  - at barriers
  - at receive
  - at flags, even at mutex

$$\frac{\text{Sequential Work}}{Max\ (\text{Work} + \text{Synch Wait Time})}$$

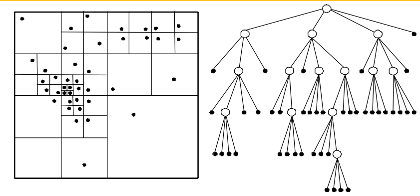
Improving Load Balance

- Decompose into more smaller tasks (>>P)
- Distribute uniformly
  - variable sized task
  - randomize
  - bin packing
  - dynamic assignment
- Schedule more carefully
  - avoid serialization
  - estimate work
  - use history info.




```
for_all i = 1 to n do
    for_all j = i to n do
        A[ i, j ] = A[i-1, j] + A[i, j-1] + ...
```


Example: Barnes-Hut

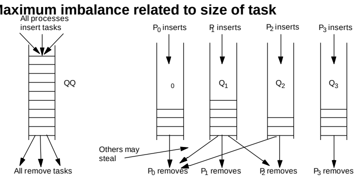


(a) The spatial domain (b) Quadtree representation

- Divide space into roughly equal # particles
- Particles close together in space should be on same processor
- Nonuniform, dynamically changing

Dynamic Scheduling with Task Queues

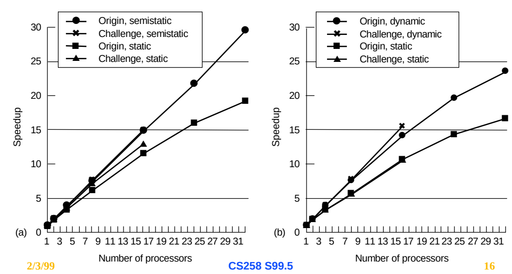
- Centralized versus distributed queues
- Task stealing with distributed queues
  - Can compromise comm and locality, and increase synchronization
  - Whom to steal from, how many tasks to steal, ...
  - Termination detection
  - Maximum imbalance related to size of task



(a) Centralized task queue (b) Distributed task queues (one per process)

Impact of Dynamic Assignment

- Barnes-Hut on SGI Origin 2000 (cache-coherent shared memory):



Self-Scheduling

```
volatile int row_index = 0;          /* shared index variable */

while (not done) {
    initialize row_index; barrier;
    while ((i = fetch_and_inc(&row_index) < n) {
        for (j = i; j < n; j++) {
            A[ i, j ] = A[i-1, j] + A[i, j-1] + ...
        }
    }
}
```


Reducing Serialization

- Careful about assignment and orchestration
  - including scheduling
- Event synchronization
  - Reduce use of conservative synchronization
    - » e.g. point-to-point instead of barriers, or granularity of pt-to-pt
  - But fine-grained synch more difficult to program, more synch ops.
- Mutual exclusion
  - Separate locks for separate data
    - » e.g. locking records in a database: lock per process, record, or field
    - » lock per task in task queue, not per queue
    - » finer grain => less contention/serialization, more space, less reuse
  - Smaller, less frequent critical sections
    - » don't do reading/testing in critical section, only modification
  - Stagger critical sections in time

NOW Handout Page 3

Impact of Efforts to Balance Load

- Parallelism Management overhead?
- Communication?
  - amount, size, frequency?
- Synchronization?
  - type? frequency?
- Opportunities for replication?
- What can architecture do?

Arch. Implications of Load Balance

- Naming
  - global position independent naming separates decomposition from layout
  - allows diverse, even dynamic assignments
- Efficient Fine-grained communication & synch
  - more, smaller
    - » msgs
    - » locks
  - point-to-point
- Automatic replication

Reducing Extra Work

- Common sources of extra work:
  - Computing a good partition
    - » e.g. partitioning in Barnes-Hut or sparse matrix
  - Using redundant computation to avoid communication
  - Task, data and process management overhead
    - » applications, languages, runtime systems, OS
  - Imposing structure on communication
    - » coalescing messages, allowing effective naming
- Architectural Implications:
  - Reduce need by making communication and orchestration efficient

$$\text{Speedup} \leq \frac{\text{Sequential Work}}{\mathit{Max}\,(\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})}$$

Reducing Inherent Communication

$$\text{Speedup} \leq \frac{\text{Sequential Work}}{\mathit{Max}\,(\text{Work} + \text{Synch Wait Time} + \text{Comm Cost})}$$

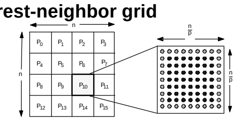
- Communication is expensive!
- Measure: *communication to computation ratio*
- Inherent communication
  - Determined by assignment of tasks to processes
  - One produces data consumed by others

=> Use algorithms that communicate less

=> Assign tasks that access same data to same process
  - same row or block to same process in each iteration

Domain Decomposition

- Works well for scientific, engineering, graphics, ... applications
- Exploits local-biased nature of physical problems
  - Information requirements often short-range
  - Or long-range but fall off with distance
- Simple example: nearest-neighbor grid computation



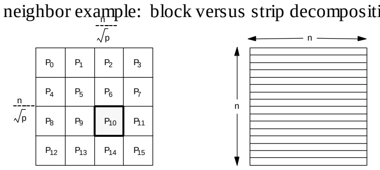
Perimeter to Area comm-to-comp ratio (area to volume in 3-d)

•Depends on n,p: decreases with n, increases with p

Domain Decomposition (contd)

Best domain decomposition depends on information requirements

Nearest neighbor example: block versus strip decomposition:

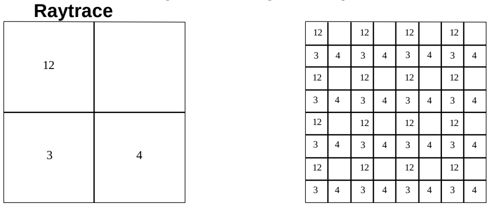


- Comm to comp: $\frac{4*\sqrt{p}}{n}$ for block, $\frac{2*p}{n}$ for strip
- Application dependent: strip may be better in other cases
  - E.g. particle flow in tunnel

NOW Handout Page 4

Relation to load balance

- Scatter Decomposition, e.g. initial partition in Raytrace



Domain decomposition   Scatter decomposition

Preserve locality in task stealing

•Steal large tasks for locality, steal from same queues, ...

Implications of Comm-to-Comp Ratio

- Architects examine application needs to see where to spend effort
  - bandwidth requirements (operations / sec)
  - latency requirements (sec/operation)
    - » time spent waiting
- Actual impact of comm. depends on structure and cost as well
- Need to keep communication balanced across processors as well

$$\text{Speedup} \leq \frac{\text{Sequential Work}}{\mathit{Max}\,(\text{Work} + \text{Synch Wait Time} + \text{Comm Cost})}$$

Structuring Communication

- Given amount of comm, goal is to reduce cost
- Cost of communication as seen by process:

$$C = f * (o + l + \frac{n_c/m}{B} + t_c - overlap)$$

» f = frequency of messages
» o = overhead per message (at both ends)
» l = network delay per message
» n_c = total data sent
» m = number of messages
» B = bandwidth along path (determined by network, NI, assist)
» t_c = cost induced by contention per message
» *overlap* = amount of latency hidden by overlap with comp. or comm.

– Portion in parentheses is cost of a message (as seen by processor)
» ignoring overlap, is *latency* of a message

–Goal: reduce terms in latency and increase overlap

Reducing Overhead

- Can reduce no. of messages ***m*** or overhead per message ***o***
- ***o*** is usually determined by hardware or system software
  - Program should try to reduce m by coalescing messages
  - More control when communication is explicit
- Coalescing data into larger messages:
  - Easy for regular, coarse-grained communication
  - Can be difficult for irregular, naturally fine-grained communication
    - » may require changes to algorithm and extra work
      - coalescing data and determining what and to whom to send
    - » will discuss more in implications for programming models later

Reducing Network Delay

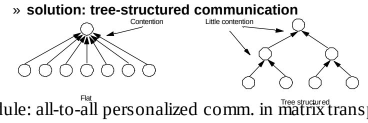
- Network delay component = $f*h*t_h$
  - » h = number of hops traversed in network
  - » t_h = link+switch latency per hop
- Reducing *f*: communicate less, or make messages larger
- Reducing *h*:
  - Map communication patterns to network topology
    - » e.g. nearest-neighbor on mesh and ring; all-to-all
  - How important is this?
    - » used to be major focus of parallel algorithms
    - » depends on no. of processors, how t_h compares with other components
    - » less important on modern machines
      - overheads, processor count, multiprogramming

Reducing Contention

- All resources have nonzero occupancy
  - Memory, communication controller, network link, etc.
  - Can only handle so many transactions per unit time
- Effects of contention:
  - Increased end-to-end cost for messages
  - Reduced available bandwidth for individual messages
  - Causes imbalances across processors
- Particularly insidious performance problem
  - Easy to ignore when programming
  - Slow down messages that don't even need that resource
    - » by causing other dependent resources to also congest
  - Effect can be devastating: *Don't flood a resource!*

Types of Contention

- **Network contention and end-point contention (hot-spots)**
- **Location and Module Hot-spots**
  - **Location: e.g. accumulating into global variable, barrier**
    - » **solution: tree-structured communication**



•Module: all-to-all personalized comm. in matrix transpose

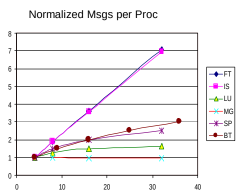
–solution: stagger access by different processors to same node temporally

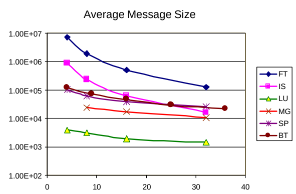
•In general, reduce burstiness; may conflict with making messages larger

Overlapping Communication

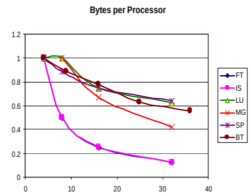
- **Cannot afford to stall for high latencies**
  - **even on uniprocessors!**
- **Overlap with computation or communication to hide latency**
- **Requires extra concurrency (*slackness*), higher bandwidth**
- **Techniques:**
  - **Prefetching**
  - **Block data transfer**
  - **Proceeding past communication**
  - **Multithreading**

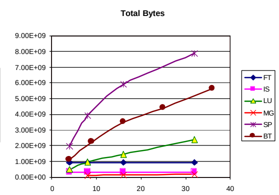
Communication Scaling (NPB2)





Communication Scaling: Volume





What is a Multiprocessor?

- **A collection of communicating processors**
  - **View taken so far**
  - **Goals: balance load, reduce inherent communication and extra work**
- **A multi-cache, multi-memory system**
  - **Role of these components essential regardless of programming model**
  - **Prog. model and comm. abstr. affect specific performance tradeoffs**

Memory-oriented View

- **Multiprocessor as Extended Memory Hierarchy**
  - » **as seen by a given processor**
- **Levels in extended hierarchy:**
  - **Registers, caches, local memory, remote memory (topology)**
  - **Glued together by communication architecture**
  - **Levels communicate at a certain granularity of data transfer**
- **Need to exploit spatial and temporal locality in hierarchy**
  - **Otherwise extra communication may also be caused**
  - **Especially important since communication is expensive**

Uniprocessor

- Performance depends heavily on memory hierarchy
- Time spent by a program

  $Time_{prog}(1) = Busy(1) + Data\ Access(1)$

  – Divide by cycles to get CPI equation
- Data access time can be reduced by:
  – Optimizing machine: bigger caches, lower latency...
  – Optimizing program: temporal and spatial locality

Extended Hierarchy

- Idealized view: local cache hierarchy + single main memory
- But reality is more complex
  – Centralized Memory: caches of other processors
  – Distributed Memory: some local, some remote; + network topology
  – Management of levels
    » caches managed by hardware
    » main memory depends on programming model
      • SAS: data movement between local and remote transparent
      • message passing: explicit
  – Levels closer to processor are lower latency and higher bandwidth
  – Improve performance through architecture or program locality
  – Tradeoff with parallelism; need good node performance and parallelism

Artifactual Communication

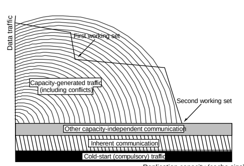
- Accesses not satisfied in local portion of memory hierachy cause communication
  – Inherent communication, implicit or explicit, causes transfers
    » determined by program
  – Artifactual communication
    » determined by program implementation and arch. interactions
    » poor allocation of data across distributed memories
    » unnecessary data in a transfer
    » unnecessary transfers due to system granularities
    » redundant communication of data
    » finite replication capacity (in cache or main memory)
  – Inherent communication assumes unlimited capacity, small transfers, perfect knowledge of what is needed.

Communication and Replication

- Comm induced by finite capacity is most fundamental artifact
  – Like cache size and miss rate or memory traffic in uniprocessors
  – Extended memory hierarchy view useful for this relationship
- View as three level hierarchy for simplicity
  – Local cache, local memory, remote memory (ignore network topology)
- Classify "misses" in "cache" at any level as for uniprocessors
    » compulsory or cold misses (no size effect)
    » capacity misses (yes)
    » conflict or collision misses (yes)
    » communication or coherence misses (no)
  – Each may be helped/hurt by large transfer granularity (spatial locality)

Working Set Perspective

•At a given level of the hierarchy (to the next further one)



– Hierarchy of working sets
– At first level cache (fully assoc, one-word block), inherent to algorithm
  » *working set curve* for program
– Traffic from any type of miss can be local or nonlocal (communication)

Orchestration for Performance

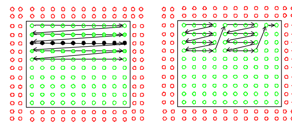
- Reducing amount of communication:
  – Inherent: change logical data sharing patterns in algorithm
  – Artifactual: exploit spatial, temporal locality in extended hierarchy
    » Techniques often similar to those on uniprocessors
- Structuring communication to reduce cost

Reducing Artifactual Communication

- Message passing model
  - Communication and replication are both explicit
  - Even artifactual communication is in explicit messages
    - » send data that is not used
- Shared address space model
  - More interesting from an architectural perspective
  - Occurs transparently due to interactions of program and system
    - » sizes and granularities in extended memory hierarchy
- Use shared address space to illustrate issues

Exploiting Temporal Locality

- Structure algorithm so working sets map well to hierarchy
  - » often techniques to reduce inherent communication do well here
  - » schedule tasks for data reuse once assigned
- Multiple data structures in same phase
  - » e.g. database records: local versus remote
- Solver example: blocking



(a) Unblocked access pattern in a sweep (b) Blocked access pattern with B = 4

•More useful when $O(n^{k+1})$ computation on $O(n^k)$ data

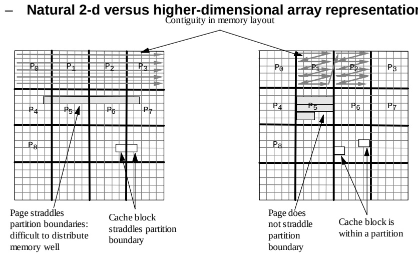
–many linear algebra computations (factorization, matrix multiply)

Exploiting Spatial Locality

- Besides capacity, granularities are important:
  - Granularity of allocation
  - Granularity of communication or data transfer
  - Granularity of coherence
- Major spatial-related causes of artifactual communication:
  - Conflict misses
  - Data distribution/layout (allocation granularity)
  - Fragmentation (communication granularity)
  - False sharing of data (coherence granularity)
- All depend on how spatial access patterns interact with data structures
  - Fix problems by modifying data structures, or layout/alignment
- Examine later in context of architectures
  - one simple example here: data distribution in SAS solver

Spatial Locality Example

- Repeated sweeps over 2-d grid, each time adding 1 to elements
- Natural 2-d versus higher-dimensional array representation



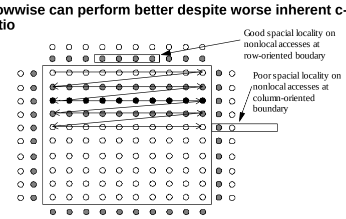
(a) Two-dimensional array (b) Four-dimensional array

Architectural Implications of Locality

- Communication abstraction that makes exploiting it easy
- For cache-coherent SAS, e.g.:
  - Size and organization of levels of memory hierarchy
    - » cost-effectiveness: caches are expensive
    - » caveats: flexibility for different and time-shared workloads
  - Replication in main memory useful? If so, how to manage?
    - » hardware, OS/runtime, program?
  - Granularities of allocation, communication, coherence (?)
    - » small granularities => high overheads, but easier to program
- Machine granularity (resource division among processors, memory...)

Tradeoffs with Inherent Communication

- Partitioning grid solver: blocks versus rows
  - Blocks still have a spatial locality problem on remote data
  - Rowwise can perform better despite worse inherent c-to-c ratio

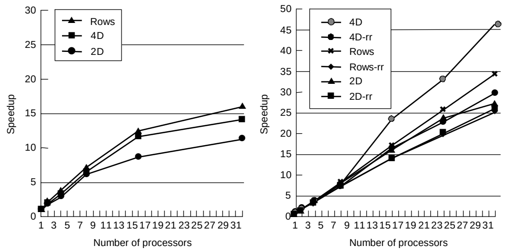


•Result depends on n and p

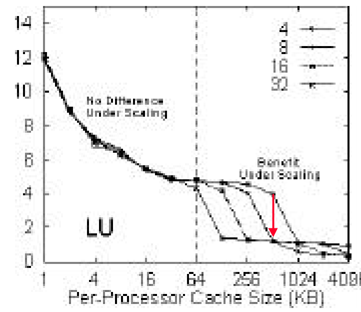
NOW Handout Page 8

Example Performance Impact

- Equation solver on SGI Origin2000

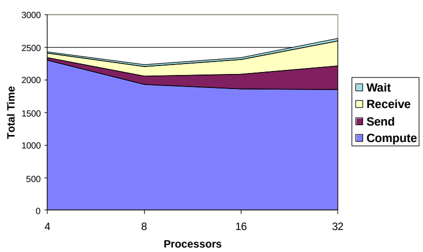


Working Sets Change with P



8-fold reduction in miss rate from 4 to 8 proc

Where the Time Goes: LU-a



Summary of Tradeoffs

- Different goals often have conflicting demands
  - Load Balance
    - » fine-grain tasks
    - » random or dynamic assignment
  - Communication
    - » usually coarse grain tasks
    - » decompose to obtain locality: not random/dynamic
  - Extra Work
    - » coarse grain tasks
    - » simple assignment
  - Communication Cost:
    - » big transfers: amortize overhead and latency
    - » small transfers: reduce contention

NOW Handout Page 9