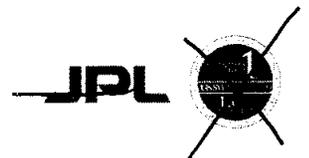


# **NavP: Structured and Multithreaded Distributed Parallel Programming**

**A Tutorial Presented to  
ACM/SIGARCH International Conference on Supercomputing (ICS'06)  
Cairns, Australia, Saturday, 7/1/06**

**Lei Pan  
Jet Propulsion Laboratory  
California Institute of Technology, USA**

**Jingling Xue  
School of Computer Science and Engineering  
University of New South Wales, Australia**

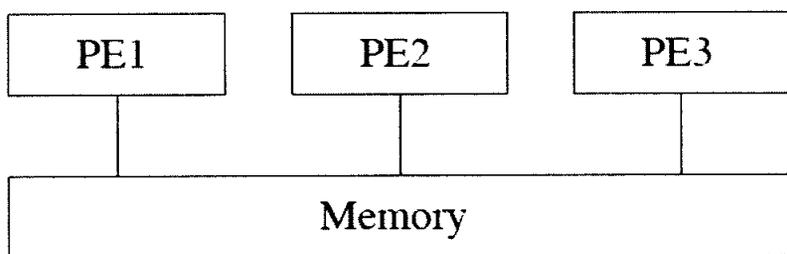


## Outline

1. **Introduction**
2. The NavP view vs. the SPMD view
3. Distributed sequential computing (DSC)
4. The methodology of NavP
5. Case Studies
6. The enabling technology of NavP
7. Comparisons, conclusions, and future work

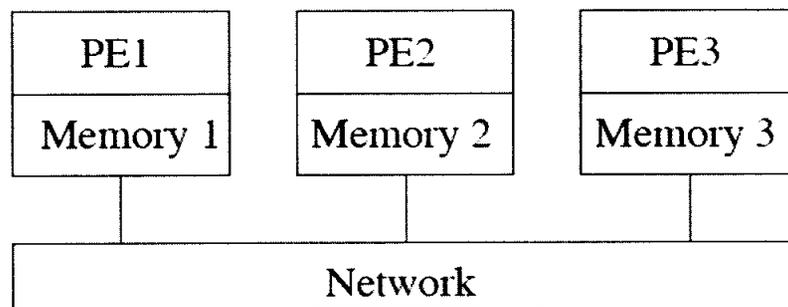
## Section 1: Introduction

- Two memory models
  - SMP (Symmetric Multi-Processors)
  - cluster (Networked Computers)
- Hybrid SMP clusters are currently popular
  - 304 out of the top 500 fastest supercomputers are clusters (as of June 2005 per top500.org)
- Cell (IBM) /Multi-core (Intel and AMD)
  - Where are the threads?



(a)

shared memory



(b)

distributed memory

## Introduction (cont'd)

- **Distribute parallel programming is a grand challenge**
  - Message passing and multithreading both needed, but very different
  - Message passing scalable for distributed memory but hard to use
  - Other proposed approaches (e.g., DSM or HPF) somewhat easier but not fast enough
  - Code ported to supercomputers does not work on PC
  - Parallelizing compilers are mostly for shared memory architectures
- **Our ability to program the clusters is lagging behind demand**
  - \$50K is about enough for a decent off-the-shelf cluster
  - \$50K is only 1/5 -- 1/3 of a man-year for a good parallel programmer
  - Parallelization of the nuclear code at DOE cost billions of dollars
  - In science and engineering, can create and collect complex data at tremendous rates, but can hardly manage and analyze the data
  - Porting sequential game to online/cluster distributed environment is expensive, and the ported code may not work well on new architectures (e.g., cell/multi-core clusters)

## Outline

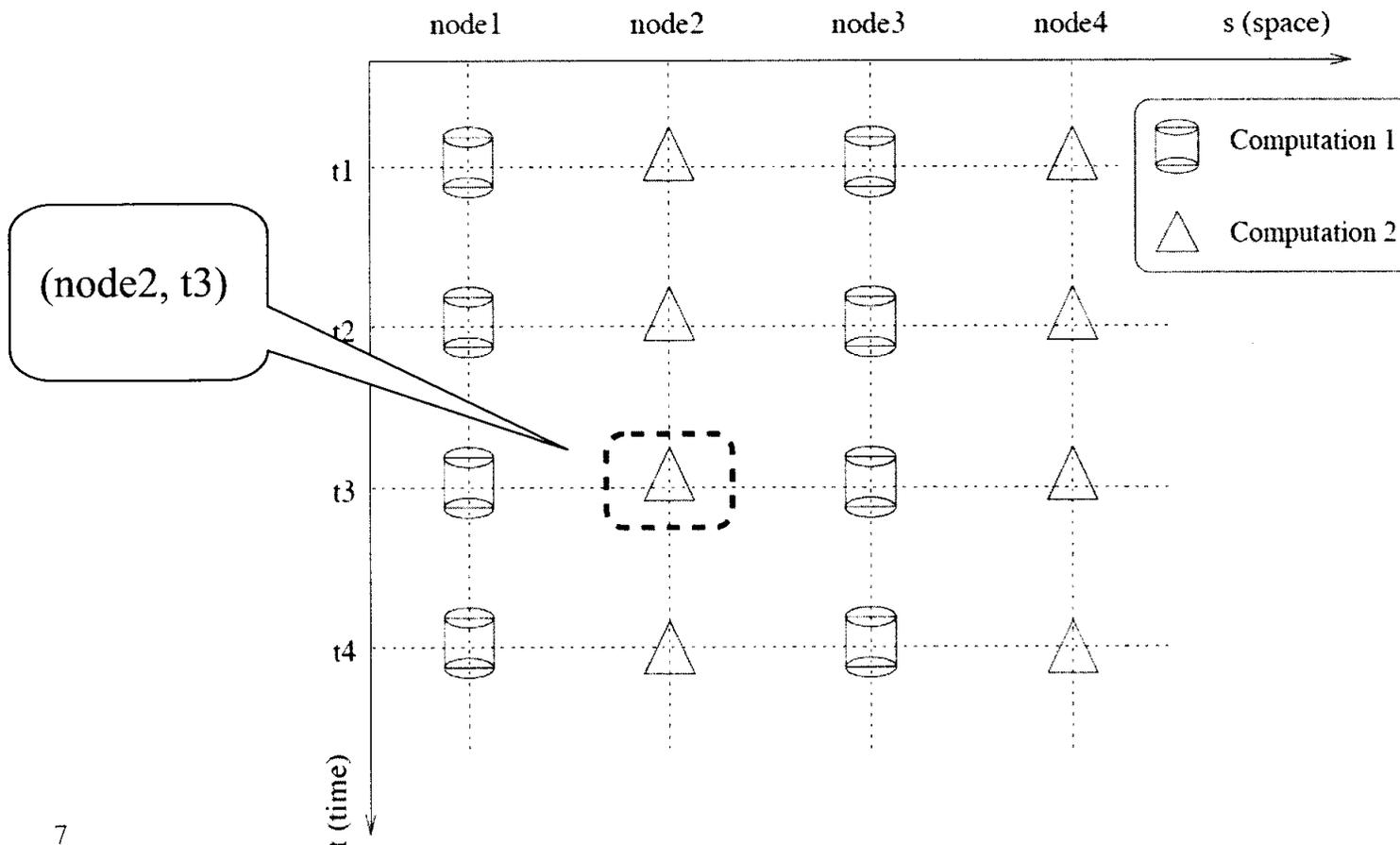
1. Introduction
2. **The NavP view vs. the SPMD view**
3. Distributed sequential computing (DSC)
4. The methodology of NavP
5. Case Studies
6. The enabling technology of NavP
7. Comparisons, conclusions, and future work

## Section 2: NavP vs. SPMD Views

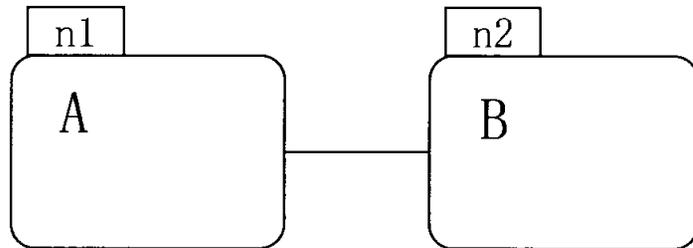
- SPMD (Single Program Multiple Data)
  - one piece of code for all processors
- NavP (Navigational Programming)
  - the programming of self-migrating computations

## Distributed Computation: A 2-D Problem

- Space for distributed computation: the Cartesian product of time and network (the net itself can be multi-dimensional)
- For simplicity, a phenomenon of distributed computation has a 2-D representation



## NavP vs. SPMD Views: Simple Example



Data distribution

```
(1) v1 = diag(A)
(2) v2 = Bv1
(3) v3 = Av2
```

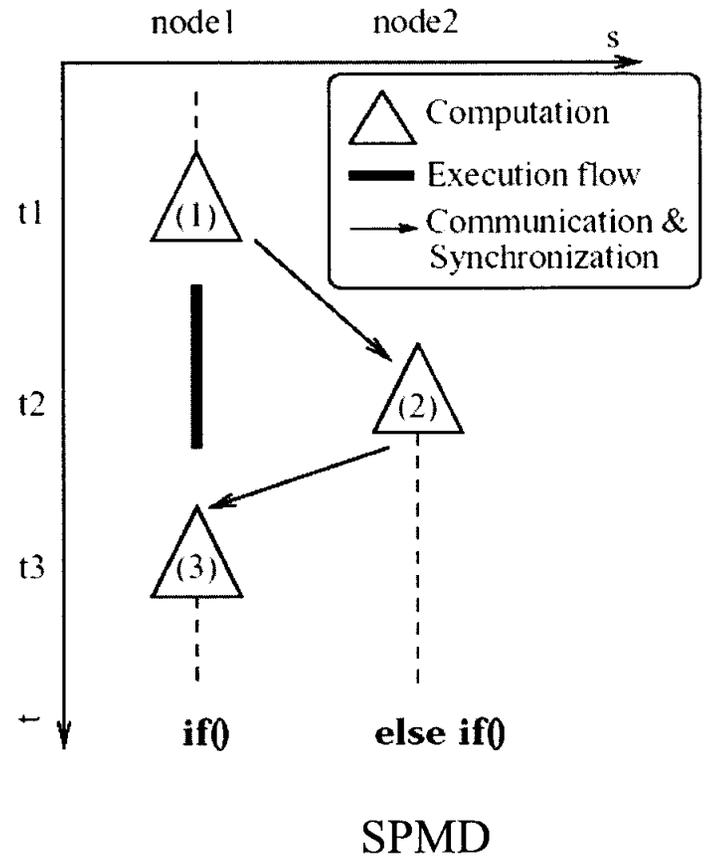
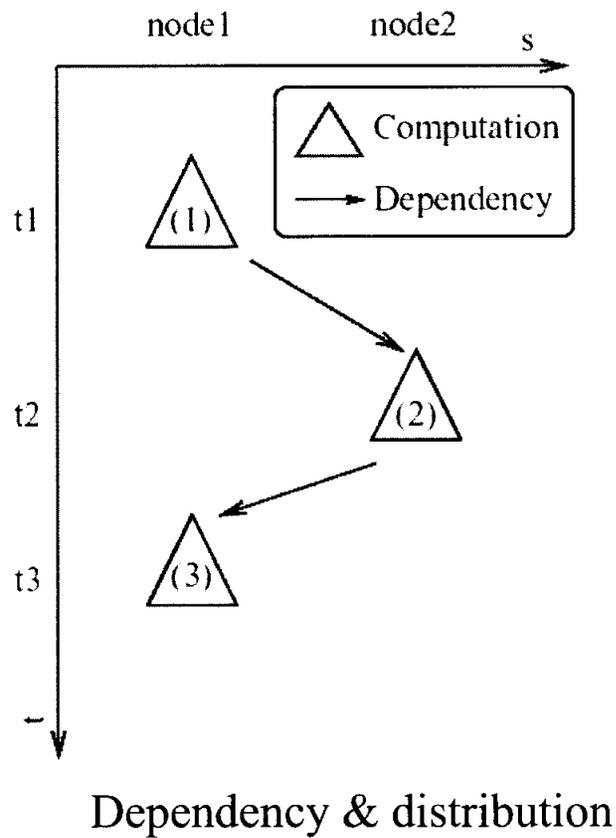
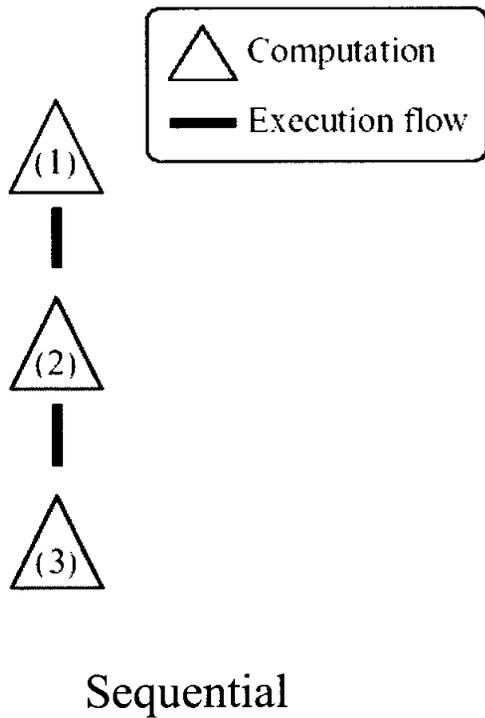
(a) Sequential

```
(0.1) If (rank = n1)
(1)   v1 = diag(A)
(1.1) send(v1, n2)
(1.2) recv(v2, n2)
(3)   v3 = Av2
(3.1) Else If (rank = n2)
(3.2) recv(v1, n1)
(2)   v2 = Bv1
(2.1) send(v2, n1)
(2.2) End If
```

(b) SPMD

# NavP vs. SPMD Views: Simple Example (cont'd)

**Line (2) no longer in between (1) and (3)!**

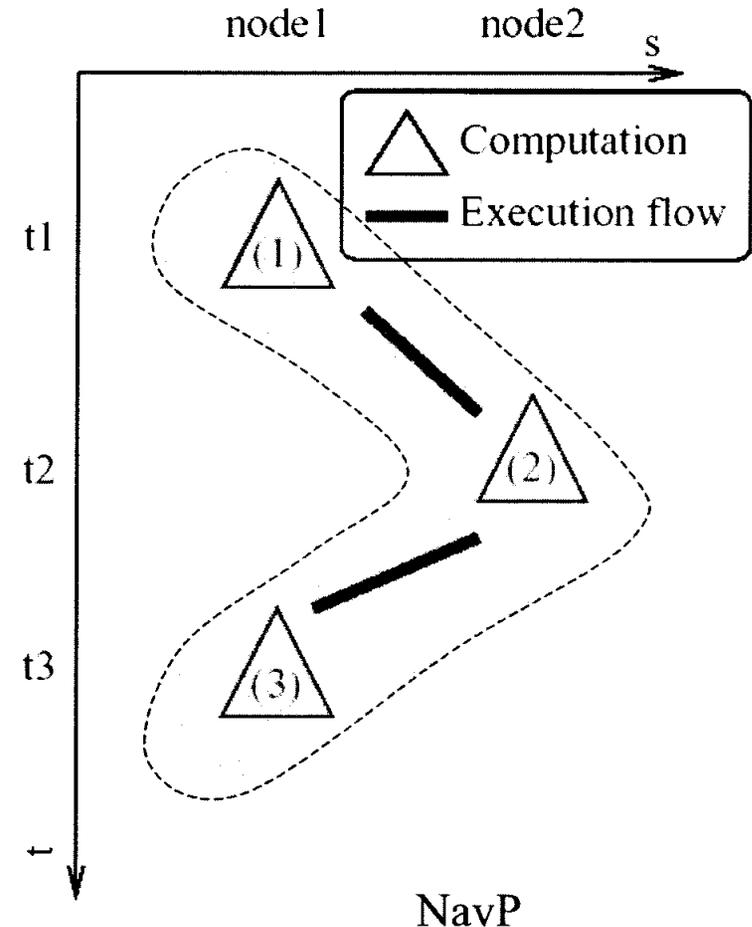


## NavP vs. SPMD Views: Simple Example (cont'd)

(1)  $v_1 = \text{diag}(A)$   
(1.1)  $\text{hop}(n_2)$   
(2)  $v_2 = Bv_1$   
(2.1)  $\text{hop}(n_1)$   
(3)  $v_3 = Av_2$

(c) NavP

- $\text{hop}(\text{dest})$  -- pauses the computation, moves the computation locus to the destination, and resumes the computation
- $v_1$  and  $v_2$  are agent variables -- variables that follow the locus of computation



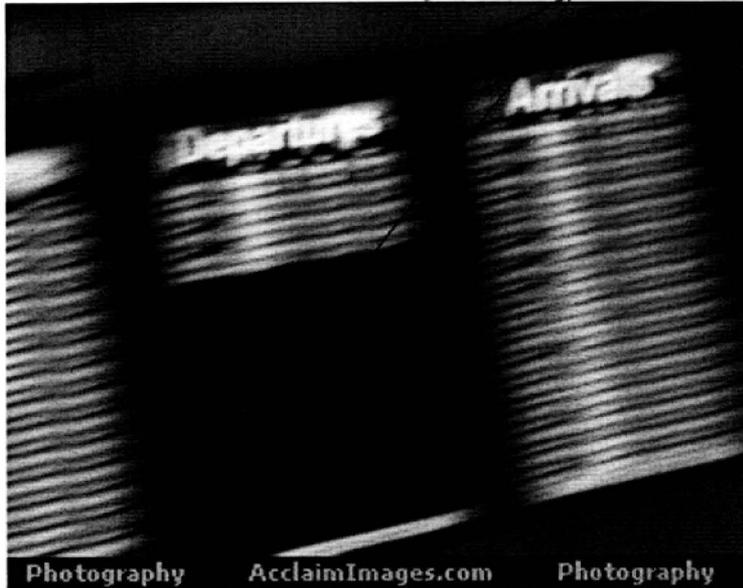
## NavP: Navigational Programming

- The programming of self-migrating computations
- Explicit navigational statements `hop()`
- Stationary data is put in “node variables”
- Carried data is put in “agent variables”
- Synchronization uses “events”
- Code does NOT move, computation does
- Overhead – book-keeping data (~200 bytes)

## NavP vs. SPMD Views

- SPMD (single program multiple data)
  - One piece of code for all processors
  - Computations are described at stationary locations
  - a.k.a. the **Eulerian** view
  - All existing models (MP, DSM, HPF) use SPMD
- NavP (navigational programming)
  - The programming of self-migrating computations
  - The description of a computation follows the migration of its locus
  - a.k.a. the **Lagrangian** view

## Analogy: Itinerary vs. Arrivals & Departures



Arrivals & departures  
(description at locations)

Itinerary  
(try inserting hops in  
between destinations)

- Arrivals & Departures: a description of train information at locations
- Itinerary: a description of trains following their movement

CUST:CJZCZJETPL AGT:DLOMW  
ITINERARY: ITIN  
RECORD LOCATOR: MH443U  
ISSUE DATE: SEP 15 2004

19 SEP 04 - SUNDAY  
TOUR  
ELECTRONIC TICKET  
COST EFFECTIVE NON REFUNDABLE  
\$100 MIN FEE FOR CHANGES/ADDITIONAL AIRFARE MAY APPLY  
NON REFUNDABLE 594.40

AIR: AMER. WEST	FLT: 630 CONFIRMED	COACH CLASS
LV: ORANGE COUNTY 1230P	EQUIP-AIRBUS A320 JET	
AR: PHOENIX 147P		NONSTOP
	AIRMILES: 338	

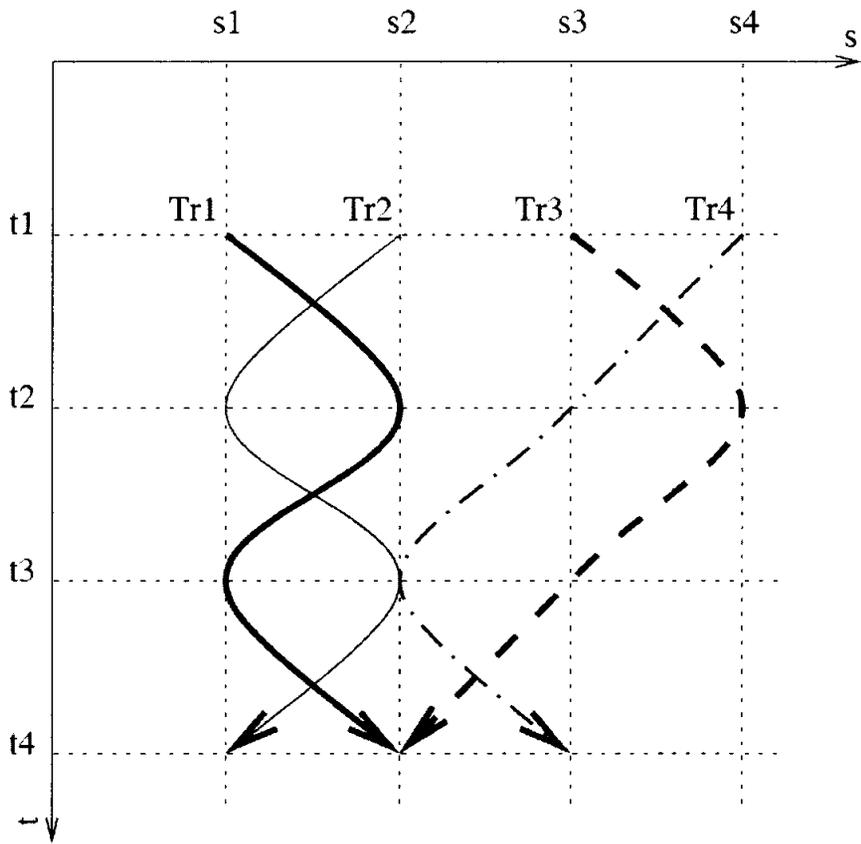
RESERVED SEATS-25D

AIR: AMER. WEST	FLT: 824 CONFIRMED	COACH CLASS
LV: PHOENIX 307P	EQUIP-AIRBUS A320 JET	
AR: BOSTON 1052P		NONSTOP
	AIRMILES: 2300	

FOOD TO PURCHASE  
RESERVED SEATS-12E  
DEPARTURE TERMINAL-4

TOUR  
NO CAR NEEDED  
MAX LODGING ALLOWED \$192 PLUS TAX-

# Analogy: Itinerary vs. Arrivals & Departures (cont'd)



	s1	s2	s3	s4
t1	Tr1	Tr2	Tr3	Tr4
t2	Tr2	Tr1	Tr4	Tr3
t3	Tr1	Tr2, Tr4	Tr3	
t4	Tr2	Tr1, Tr3	Tr4	

(a) Arrivals & Departures

	Tr1	Tr2	Tr3	Tr4
t1	s1	s2	s3	s4
t2	s2	s1	s4	s3
t3	s1	s2	s3	s2
t4	s2	s1	s2	s3

(b) Itinerary

- The information provided by one view can be presented in another
- A view is good for a purpose (taxi driver vs. traveler); SPMD is good for client/server



## Analogy: Describing a girl on a train

- How do you want to describe a girl sitting on a train?
  - Hold a video camcorder and sit next to her (**Lagrangian**)
  - Station a line of people along the railway to take snapshots of her as she passes them in the train, and assemble the pictures into a video later (**Eulerian**)



## Structured Distributed Programming: Uniprocessor Special Case

- Send/Recv are **goto** in distributed programming

```
(0.1) If (rank = n1)
(1)   v1 = diag(A)
(1.1) send(v1, n2)
(1.2) recv(v2, n2)
(3)   v3 = Av2
(3.1) Else If (rank = n2)
(3.2)  recv(v1, n1)
(2)   v2 = Bv1
(2.1) send(v2, n1)
(2.2) End If
```

(a) SPMD

```
(0.1)
(1)   v1 = diag(A)
(1.1) goto L1
(1.2) L2:
(3)   v3 = Av2
(3.1)
(3.2) L1:
(2)   v2 = Bv1
(2.1) goto L2
(2.2)
```

(b) SPSD

## The Software Crisis in the 1960s

- Goto Statement Considered Harmful
  - by Edsger W. Dijkstra, Communications of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148
  - Suggestion: Abolish goto and use structured programming
- Send-recv considered harmful: Myths and realities of message passing
  - by Sergei Gorlatch, ACM Transactions on Programming Languages and Systems, Vol. 26 , Issue 1, January 2004
  - Suggestion: Use collective operations, e.g., broadcast, reduction, etc. to replace send and recv
- We suggest to use `hop ()`

## Structured Distributed Programming (cont'd)

```
(0.1) If (rank = n1)
(1)    v1 = diag(A)
(1.1)  send(v1, n2)
(1.2)  End If

(1.3)  If (rank = n2)
(1.4)  recv(v1, n1)
(2)    v2 = Bv1
(2.1)  send(v2, n1)
(2.2)  End If

(2.3)  If (rank = n1)
(2.4)  recv(v2, n2)
(3)    v3 = Av2
(3.1)  End If
```

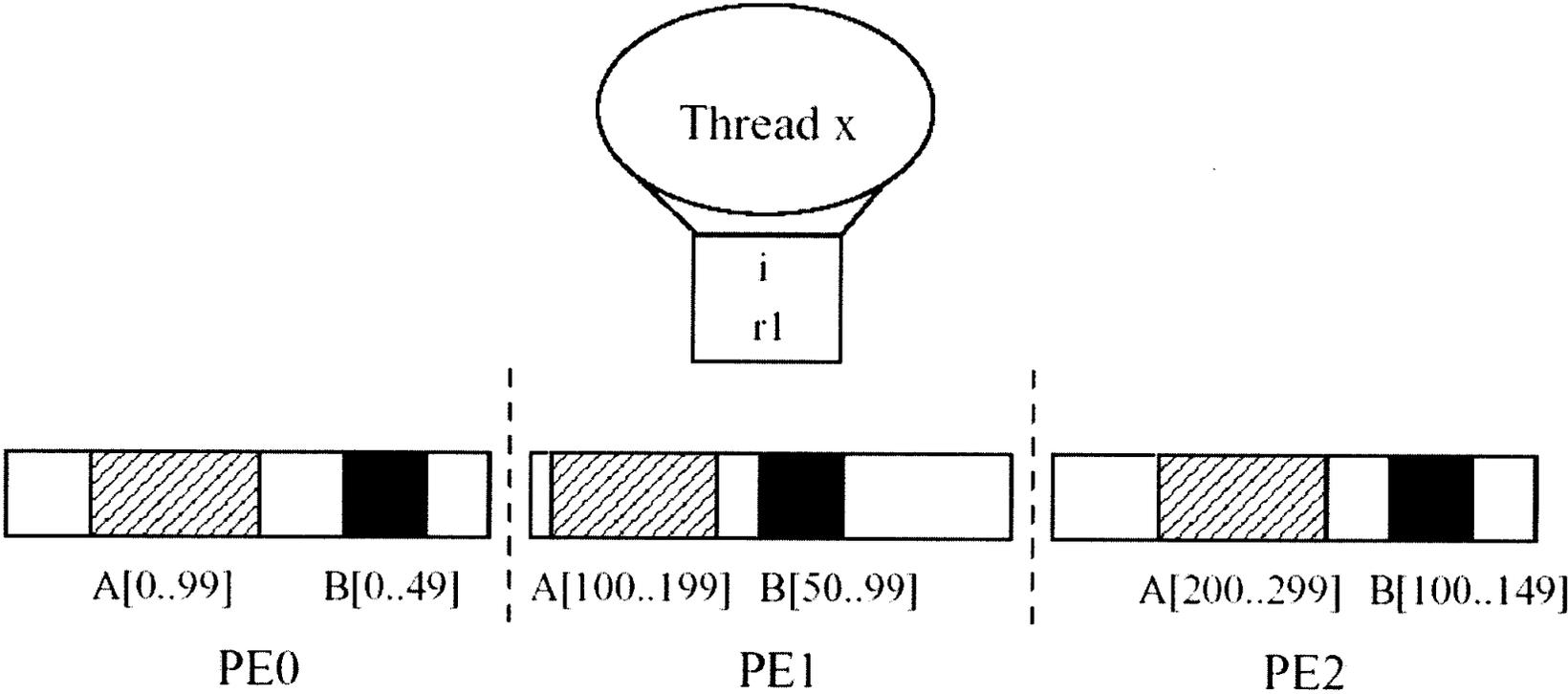
- Structured Distributed Programming with SPMD problematic, unless:
  - Manage to have corresponding `recv()` follow next to the `send()`
  - As a special case, use collective operations, e.g., broadcast, reduction, gather
- NavP preserves **Algorithmic Integrity**:
  - `Hop()` does not change the successive (in time) action as described by the original successive (in text space) action descriptions, even though the spatial location of the successive (in time) action is changed

## NavP: Shared Variable Programming Beyond Shared Memory

- Shared Variable Programming
  - Inter-process communication and synchronization managed through variables to which two or more processes have access
- SV Programming attractive because reading and writing remote memory with familiar assignment statements
- DSM uses SV Programming

# NavP: Shared Variable Programming Beyond Shared Memory

- SV Programming beyond shared memory
  - Algorithmic Integrity
- Global view of the distributed data
  - Data structure Integrity



# NavP: Shared Variable Programming Beyond Shared Memory

## ■ Node vs. agent variables

	non-distributed	distributed
non-shared	(none)	agent variable
shared	node variable	DSV

- DSV: Distributed Shared Variable
- A compound navigational statement

```
load temporary data to agent vars  
hop(DEST)  
unload temporary data from agent vars
```

## Summary: NavP vs. SPMD Views

- Introduced the NavP view for distributed computing (a.k.a. the Lagrangian view in fluid dynamics)
- Distributed computation is fundamentally Lagrangian
- NavP is the programming of self-migrating computations
- Send-recv as harmful as goto; use `hop()`
- NavP preserves Algorithmic Integrity
- Computation mobility (defined later) enables shared variable programming beyond shared memory

## Outline

1. Introduction
2. The NavP view vs. the SPMD view
3. **Distributed sequential computing (DSC)**
4. The methodology of NavP
5. Case Studies
6. The enabling technology of NavP
7. Comparisons, conclusions, and future work

## Section 3: Distributed Sequential Computing (DSC)

- Computing with distributed data using a single locus of computation
- But a cluster computer is for parallel computing?
  - Yes, parallelism is considered later
  - DSC is by itself useful
  - DSC is a fundamental composing element of NavP: a parallel program can be composed from pipelined DSC threads

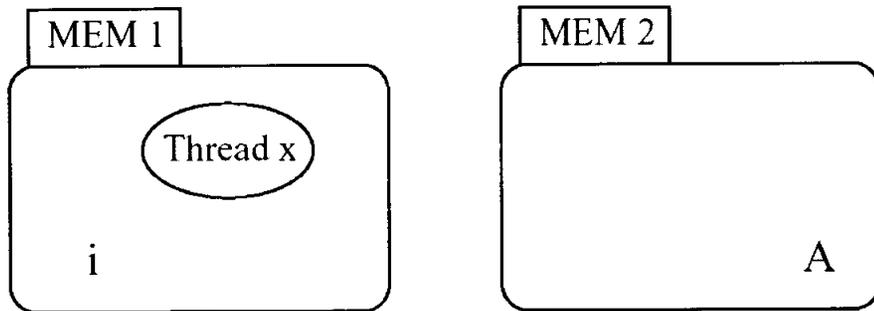
## DSC: why?

- Only few algorithms can be perfectly parallelized (embarrassingly parallel); sequential portions are unavoidable
- Granularity considerations
- Re-implementing parallel code can be a major effort, while distributed sequential computing can handle large problems
- Insight into distributed programming

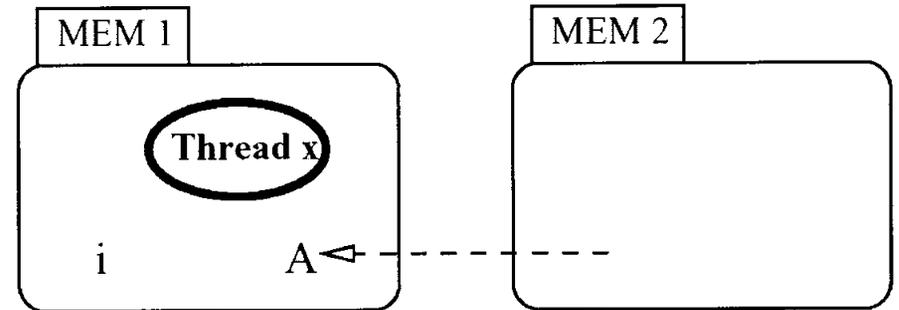
## Computation Mobility

- The ability for the locus of computation to migrate across distributed memories and continue the computation as it meets the required data

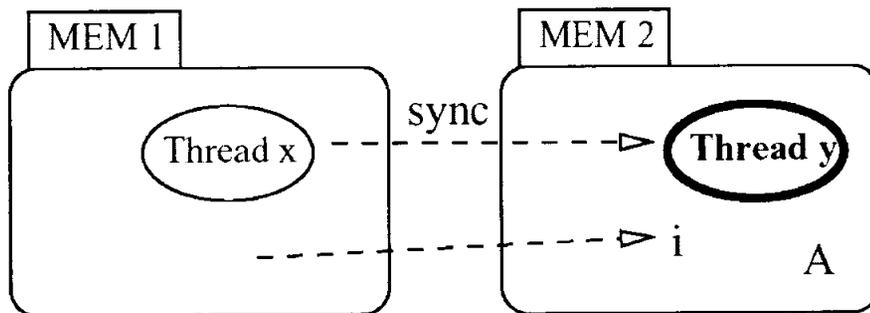
# DSC: Data and Computation Rendezvous



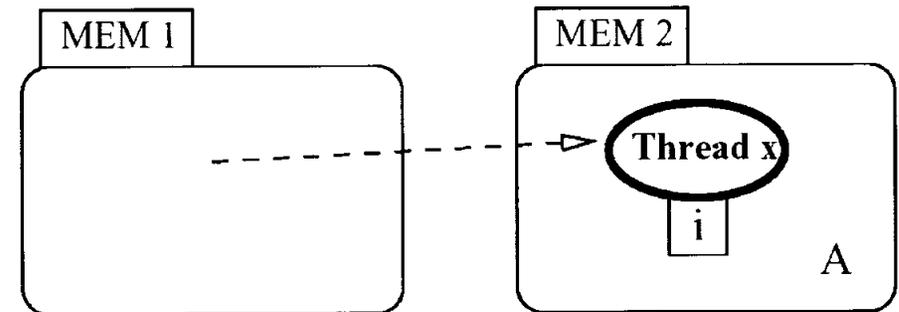
(a) The Problem



(b) DSM



(c) MP



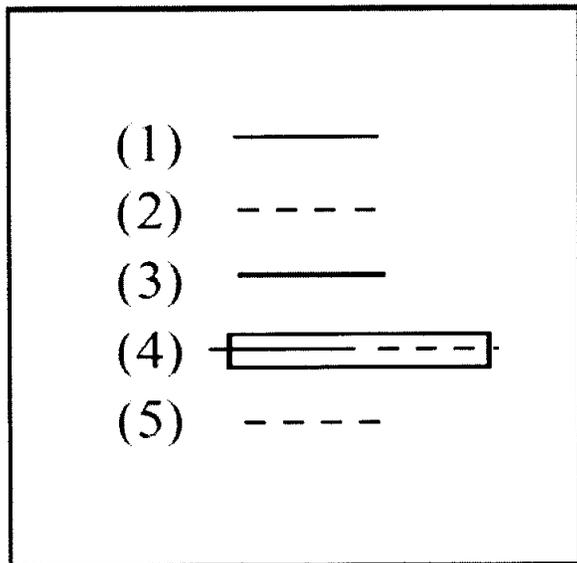
(d) DSC using computation mobility

## An Anatomy of MP and DSM

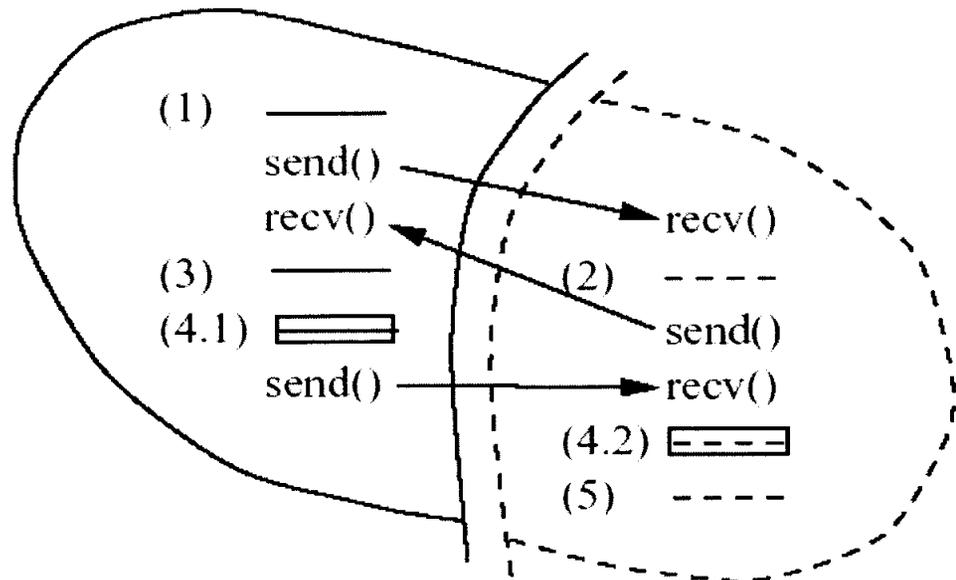
- MP code efficient in communication
- MP implementation restructures the original code
  - With MP, partitioning data means restructuring code
- DSM code the same as the original code
- Severe performance penalty with DSM
  - large data is moved to meet with small data
- Try something new?

## DBlock Resolution and DBlock Analysis

- **DBlock: Distributed code building Block**
  - Any code block containing a DBlock is a DBlock
- **DBlock Resolution: Make the data-computation rendezvous happen**
- **DBlock Analysis: Find an efficient rendezvous**



(a)  
A DBlock  $\mathcal{B}_T(\mathcal{D})$



(b)  
And its data

## The Principle of Pivot-computes

- **Principle of Pivot-computes:** The computation of a DBlock should take place on the node that hosts the largest piece of the distributed data that the DBlock accesses
- That node is called a **pivot node**

## DSC Program Transformation

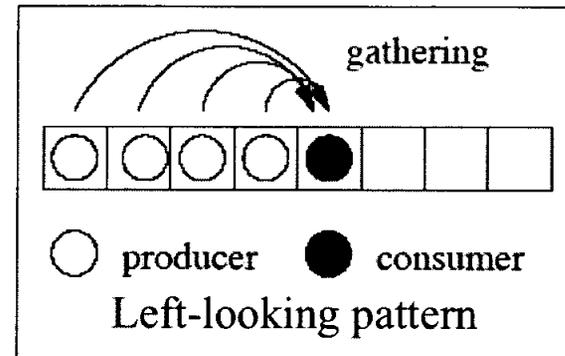
- How to implement a distributed program from a sequential program?
  - Conduct DBlock analysis to determine which DBlocks to resolve (granularity level matters)
  - Resolve those chosen DBlocks following the principle of pivot-computes
- If you do not want to restructure the original sequential code, resolve the DBlocks using computation mobility
  - Otherwise you handcraft computation mobility to follow pivot-computes

# DSC Simple Example: A Left Looking Algorithm

```

do j = 2 to n
  do i = 1 to j-1
    a[j] = (a[j]+a[i])*j/(j+i)
  end do /* computations not communitive, nor associative */

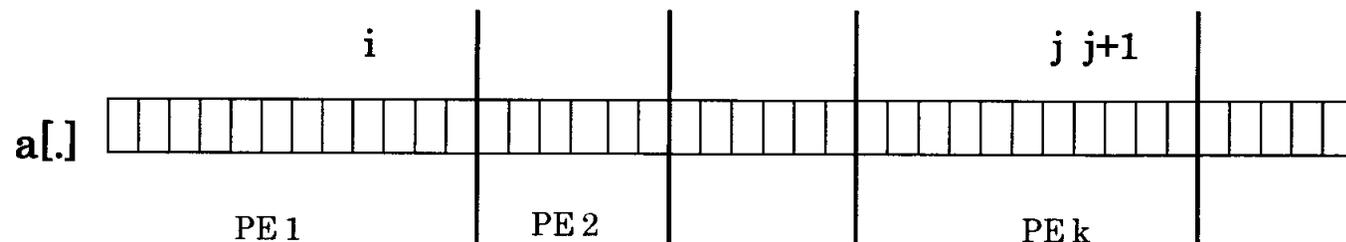
```



```

  a[j] = a[j]/j
end do

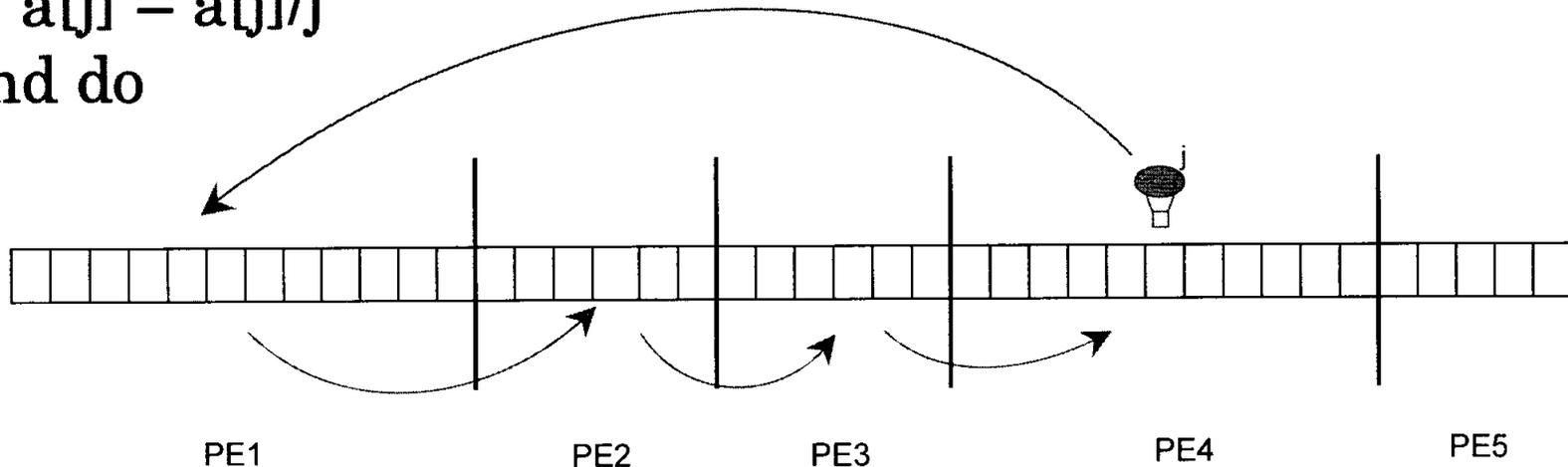
```



## DSC Simple Example: A Left Looking Algorithm (cont'd)

```
do j = 2 to n
  hop(node[j]); mx = a[j]
  do i = 1 to j-1
    hop(node[i])
    mx = (mx+a[i])*j/(j+i)
  end do
  hop(node[j]); a[j] = mx
  a[j] = a[j]/j
end do
```

- i, j, mx are “agent variables”
- a[.] is a “node variable”
- Most hops are no-ops

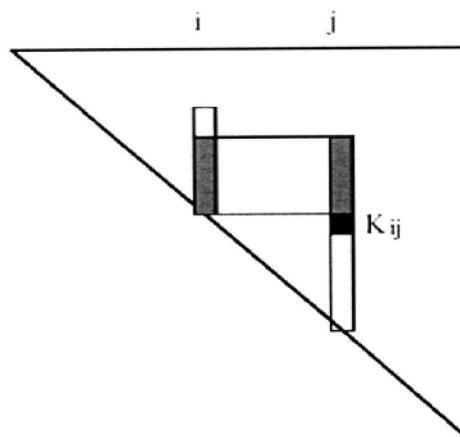


## DSC Real Example: Crout Factorization

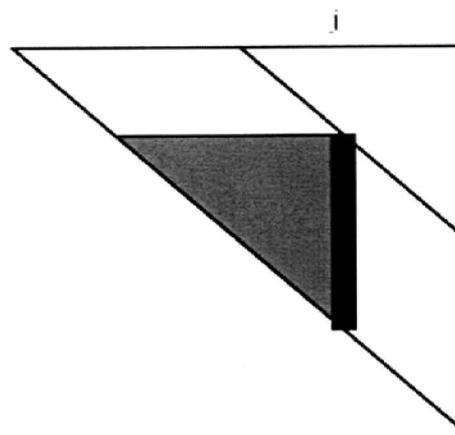
- Eliminates page faults by distributing the locus of computation on multiple nodes
- Eliminates disk paging and replace it by inexpensive and less frequent hops
- A competing approach: paging to remote memory
  - “Adaptive and Reliable Paging to Remote Main Memory,” J. of Para. and Dist. Comp., 1999
  - Violates pivot-computes; moves more data than needed
  - A special case of DSM
  - Does not reduce the number of page faults. Only improves the service time for each page fault.
  - Not a scalable solution; may cause remote memory thrashing

## DSC Real Example: Crout Factorization (cont'd)

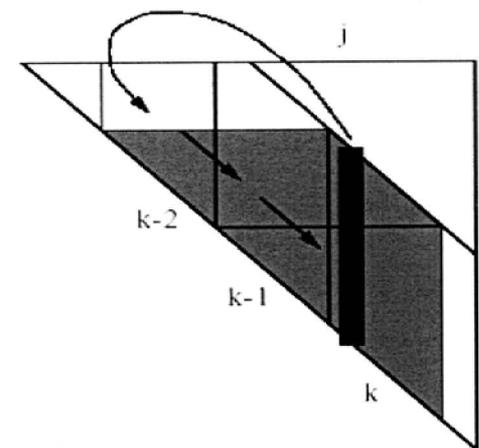
- What if the working set is too big for a computer?
  - Partition the working set
  - Distribute the working set
  - Have computation approach the partitions



Data for  $K_{ij}$

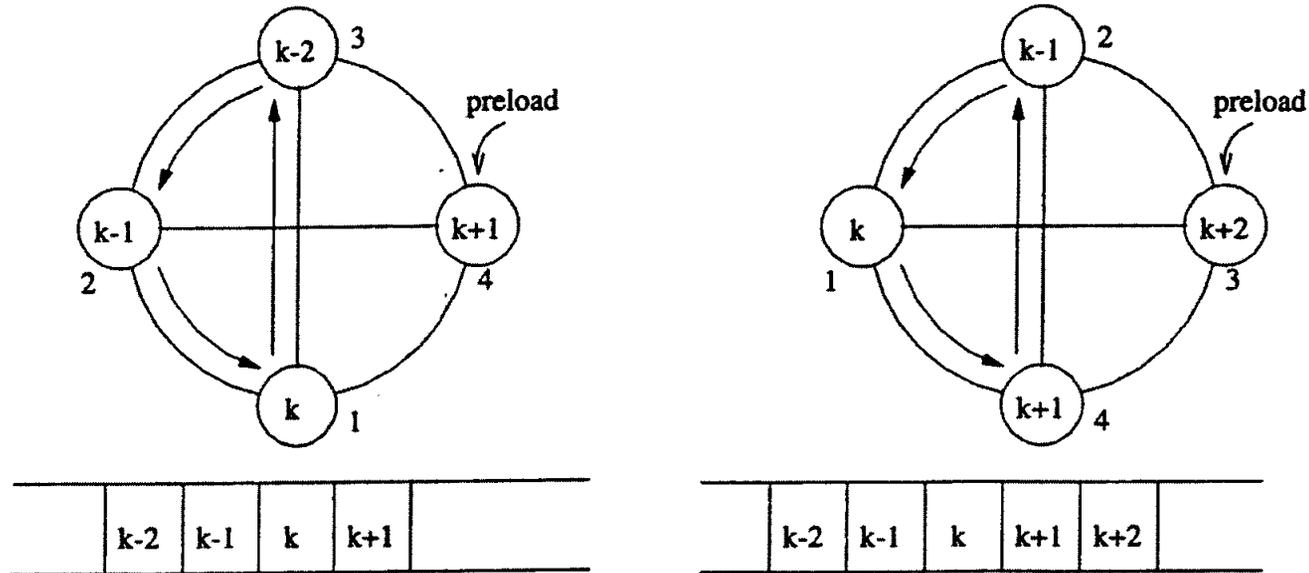


Data for column  $j$



col  $j$  put in agent variable  
and go to meet the blocks

## DSC Real Example: Crout Factorization (cont'd)



- Need enough collective memory to host working set (not entire matrix)
- Use an additional PE to pre- and post-fetch
- Use 4 workstations to solve a problem of the size of 35 machines' collective memory

## DSC Real Example: Crout Factorization (cont'd)

**DBlocks**

```

(1) For j = 1 to N
(2)   For i = 2 to j - 1
(3)      $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$ 
(4)   End For

(5)   For i = 1 to j - 1
(6)      $T \leftarrow K_{ij}$ 
(7)      $K_{ij} \leftarrow \frac{T}{K_{ii}}$ 
(8)      $K_{jj} \leftarrow K_{jj} - TK_{ij}$ 
(9)   End For
(10) End For
    
```

Sequential Crout

```

(1)   For j = 1 to N
(1.1)   load column j
(2)     For i = 2 to j - 1
(2.1)     hop to column i
(2.2)     load  $K_{ii}$ 
(3)        $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$ 
(4)     End For

(4.1)   hop to column j
(4.2)   unload column j

(5)     For i = 1 to j - 1
(6)        $T \leftarrow K_{ij}$ 
(7)        $K_{ij} \leftarrow \frac{T}{K_{ii}}$ 
(8)        $K_{jj} \leftarrow K_{jj} - TK_{ij}$ 
(9)     End For
(10)   End For
    
```

DSC Crout w/o fetching

(1) *For*  $j = 1..N$

(2) *For*  $i = 2..j - 1$

(3)  $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$

(4) *End For*

(5) *For*  $i = 1..j - 1$

(6)  $T \leftarrow K_{ij}$

(7)  $K_{ij} \leftarrow \frac{T}{K_{ii}}$

(8)  $K_{jj} \leftarrow K_{jj} - TK_{ij}$

(9) *End For*

(10) *End For*

(1) *For*  $j = 1..N$

(1.1) *hop*(*node*[*j*])

(1.2) *If* (*hopped across node*)

(1.3) *inject*(*WR*(*j*))

(1.4) *waitEvent*(*IO*<sub>*b*</sub>(*j*))

(1.5) *End If*

(1.6) *load*(*column j*)

(2) *For*  $i = 2..j - 1$

(2.1) *hop*(*node*[*i*])

(2.2) *load*(*K*<sub>*ii*</sub>)

(3)  $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$

(4) *End For*

(4.1) *hop*(*node*[*j*])

(4.2) *unload*(*column j*)

(5) *For*  $i = 1..j - 1$

(6)  $T \leftarrow K_{ij}$

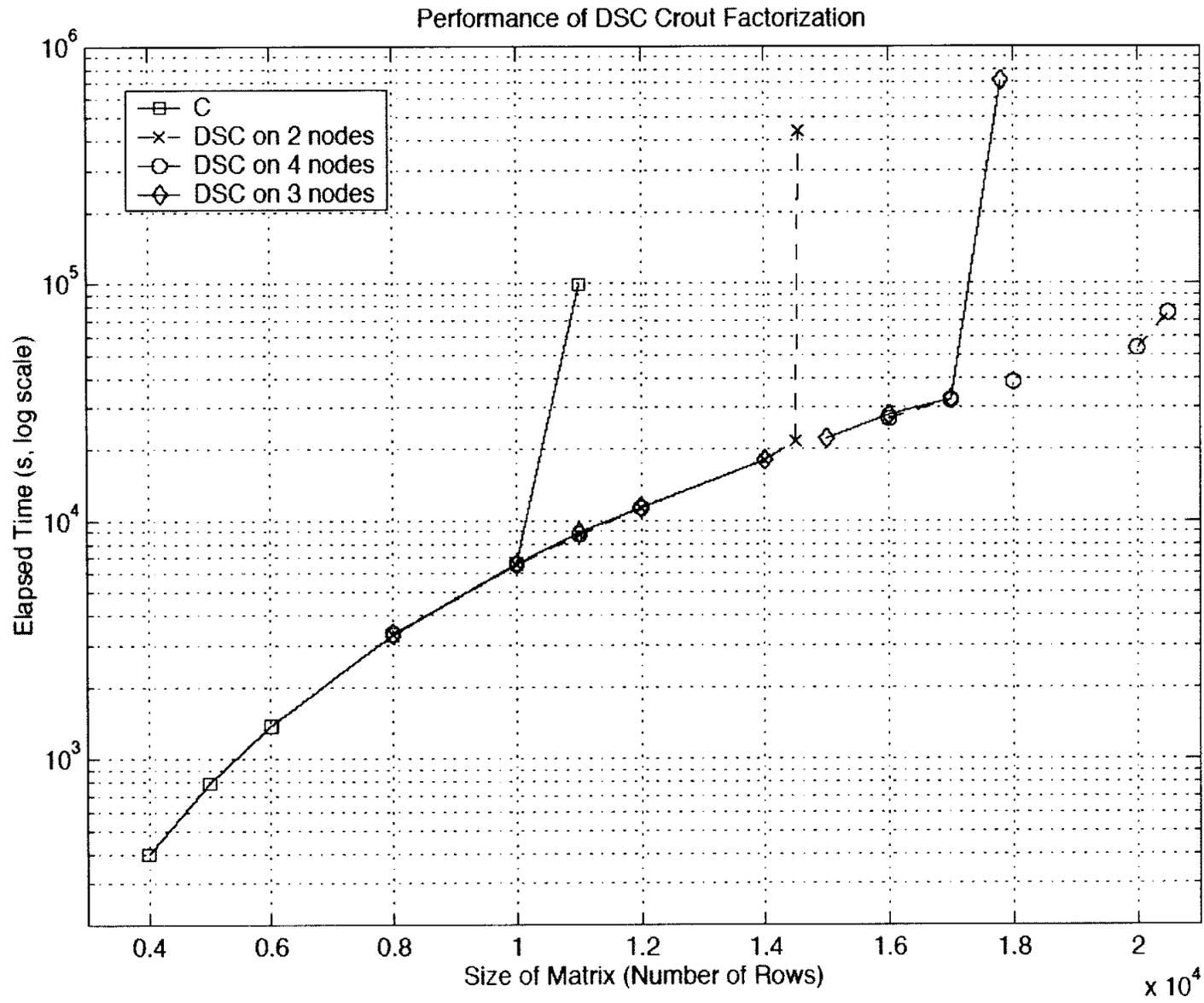
(7)  $K_{ij} \leftarrow \frac{T}{K_{ii}}$

(8)  $K_{jj} \leftarrow K_{jj} - TK_{ij}$

(9) *End For*

(10) *End For*

# DSC Real Example: Crout Factorization (cont'd)



(1) <i>For</i> $k = 1 .. P$	(17) <i>Recv</i> ( <i>col</i> $j, k$ )
(2) <i>If</i> $\mu == k$	(6') <i>For</i> $i = I_{k-2} .. I_{k-1} - 1$
(3) <i>For</i> $j = I_k .. I_{k+1} - 1$	(7') $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$
(4) <i>Send</i> ( <i>col</i> $j, k - 2$ )	(8') <i>End For</i>
(5) <i>Recv</i> ( <i>col</i> $j, \{K_{dd}\}, k - 1$ )	(18) <i>Send</i> ( <i>col</i> $j, \{K_{dd}\}, k - 1$ )
(6) <i>For</i> $i = I_k .. j - 1$	(19) <i>End For</i>
(7) $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$	(20) <i>Else If</i> $\mu == k - 1$
(8) <i>End For</i>	(21) <i>For</i> $m = I_k .. I_{k+1} - 1$
(9) <i>For</i> $i = 1 .. j - 1$	(22) <i>Recv</i> ( <i>col</i> $j, \{K_{dd}\}, k - 2$ )
(10) $T \leftarrow K_{ij}$	(6'') <i>For</i> $i = I_{k-1} .. I_k - 1$
(11) $K_{ij} \leftarrow \frac{T}{K_{ii}}$	(7'') $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$
(12) $K_{jj} \leftarrow K_{jj} - TK_{ij}$	(8'') <i>End For</i>
(13) <i>End For</i>	(23) <i>Send</i> ( <i>col</i> $j, \{K_{dd}\}, k$ )
(14) <i>End For</i>	(24) <i>End For</i>
(15) <i>Else If</i> $\mu == k - 2$	(25) <i>End If</i>
(16) <i>For</i> $m = I_k .. I_{k+1} - 1$	(26) <i>End For</i>

- 1). Both loops broken;
- 2). if()/else if() for different nodes.

**Message Passing Crout**  
(code for fetching not shown)

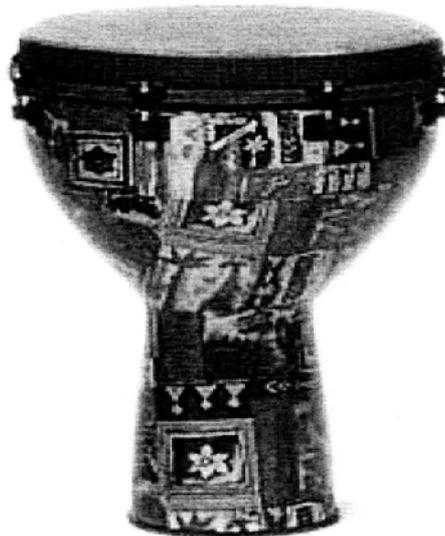
## Analogy: Drums or Drummer?

- Programming the drums?

Vibrate  
0.4 sec  
Send

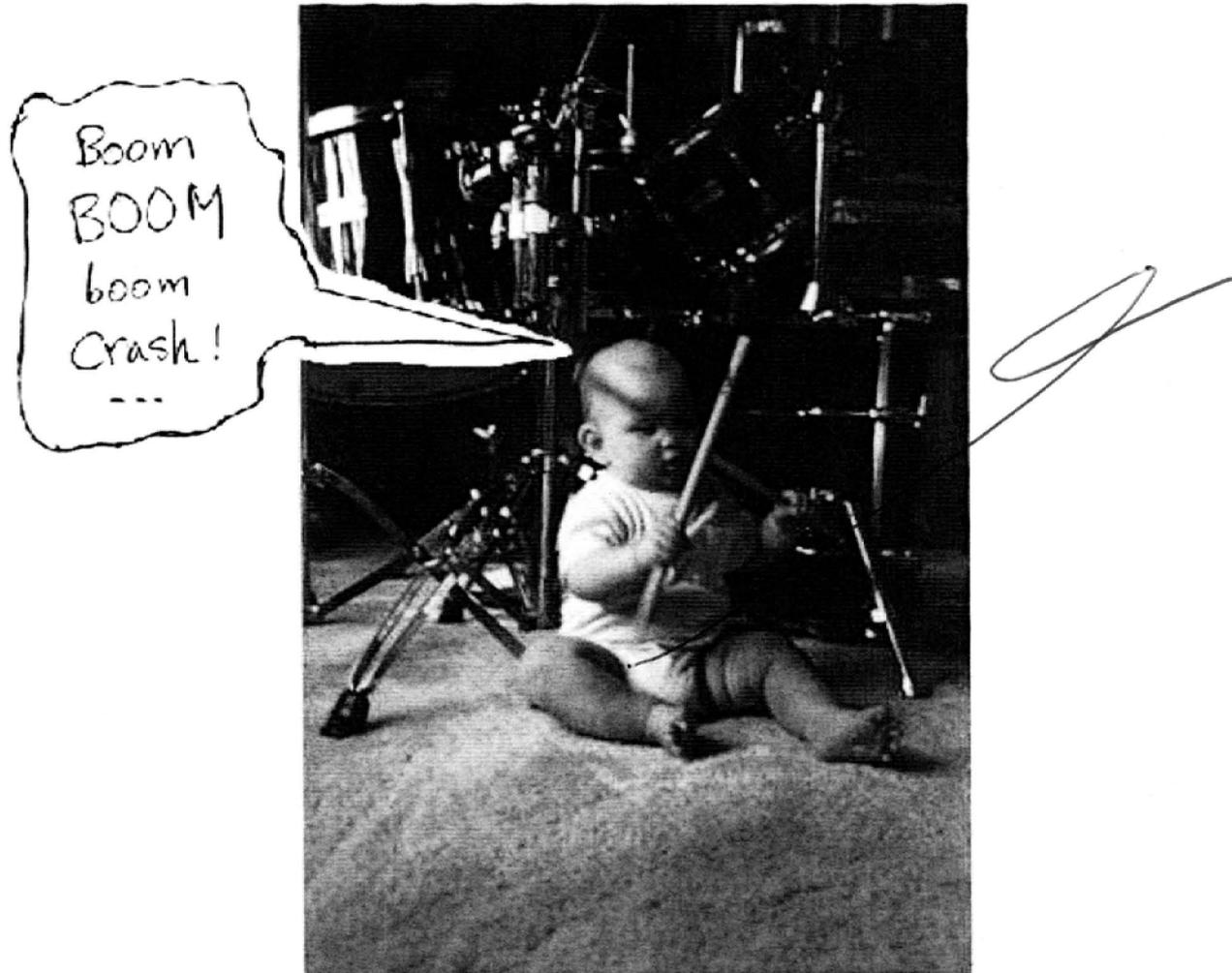
Receive  
Vibrate  
0.6 sec  
Send

Receive  
Vibrate  
0.2 sec  
Send



## Analogy: Drums or Drummer? (cont'd)

- Or the drummer?



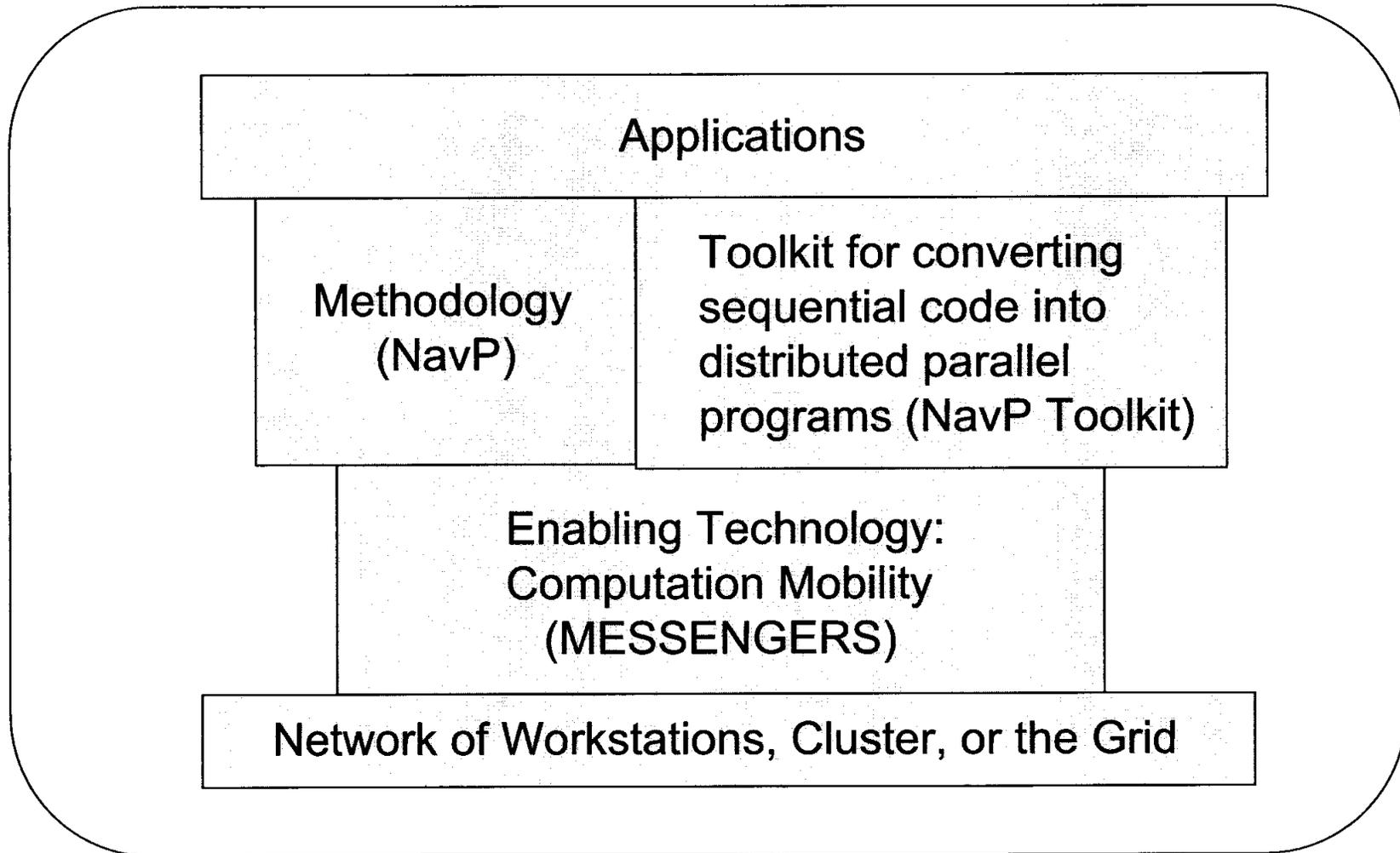
## Summary: Distributed Sequential Computing

- Introduced DSC: Distributed Sequential Computing
- Introduced Computation Mobility
- Introduced DBlock and DBlock analysis
- Introduced the Principle of Pivot-computes
- Used simple as well as real-world examples to show why DSC using NavP is efficient, scalable, and easy
- MP is not a good fit for DSC

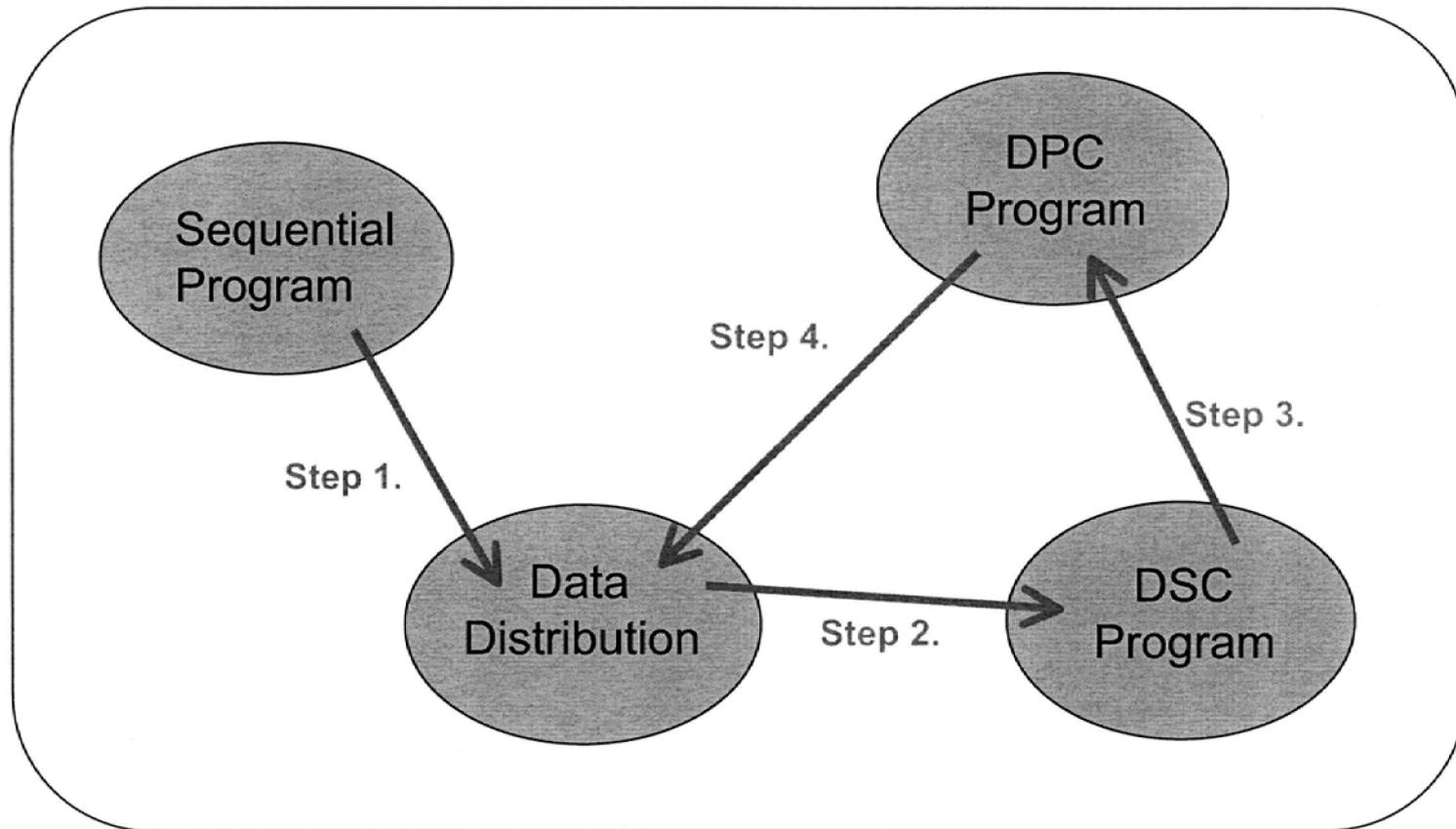
## Outline

1. Introduction
2. The NavP view vs. the SPMD view
3. Distributed sequential computing (DSC)
4. **The methodology of NavP**
5. Case Studies
6. The enabling technology of NavP
7. Comparisons, conclusions, and future work

## The NavP Building Blocks



# The NavP Steps



## The NavP Steps

### 1. Data distribution

- Affinity graph by code instrumenting
- Partition the graph to get data mapping
- Distribute data to DSV

### 2. Distributed Sequential Computing (DSC)

- Insert migration statements (**hop(.)**)
- Small data to meet with large data
- Computation made coarse grained

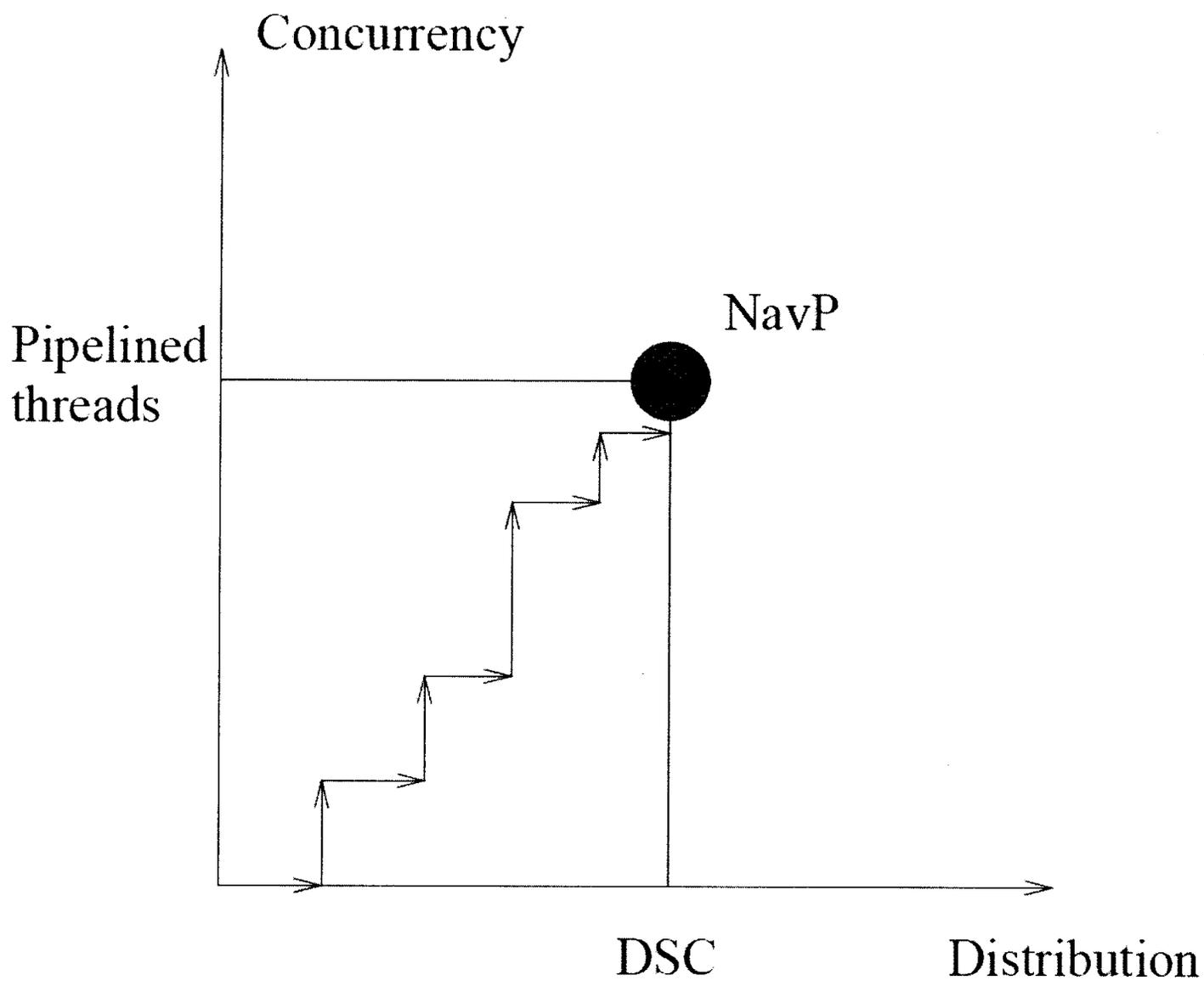
### 3. Distributed Parallel Computing (DPC)

- Decompose into and pipeline DSC threads
- Insert **signalEvent()** and **waitEvent()**

### 4. Loop back for feedback and refinement

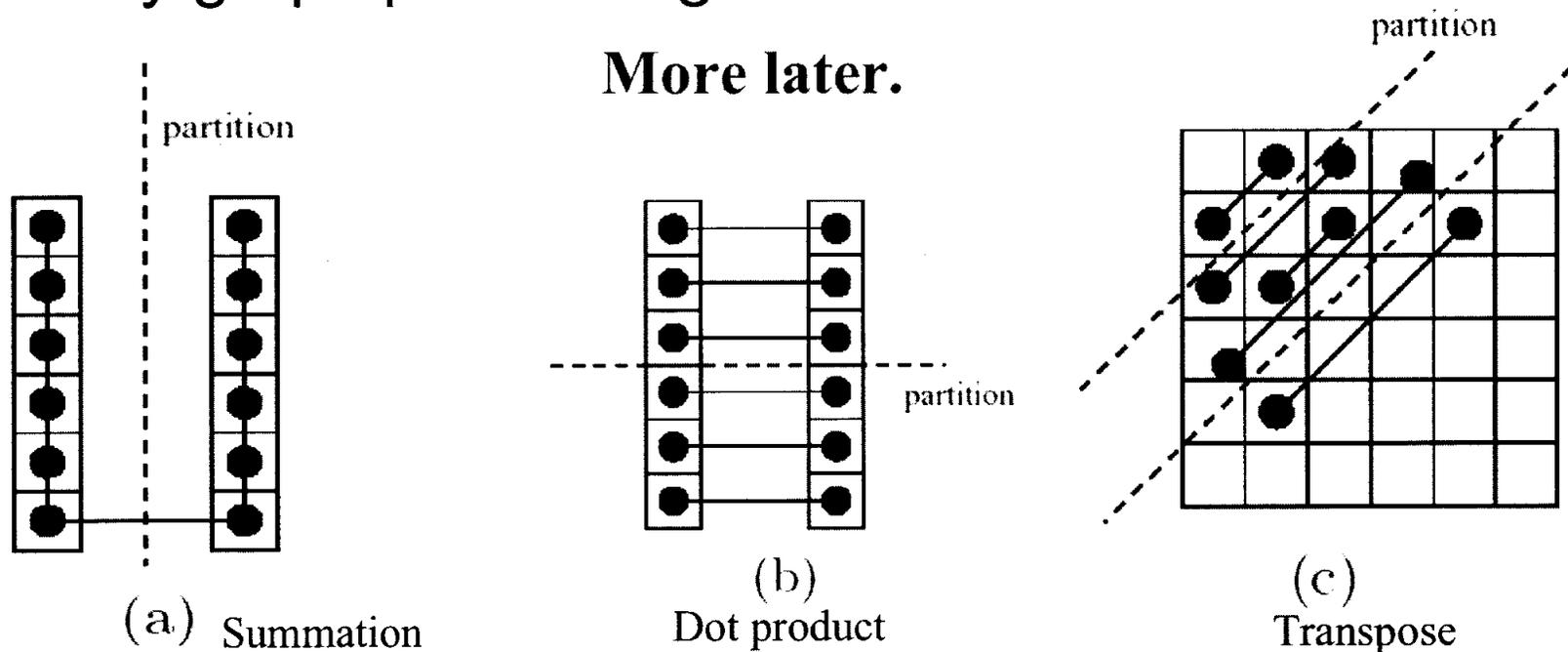
- Redistribute data to find a balanced point between degree of parallelism and cost of communication

# The NavP Steps Can Be Applied Repeatedly or Hierarchically



## Step 1: Data Distribution

- Affinity graph (sequential program “instrumentation” - obtain data access pattern info. from execution)
  - In contrast to using the dual of a mesh
- **A**(affinity), **C**(continuity), **R**(regularity) edges in affinity graph
- *k*-way graph partitioning

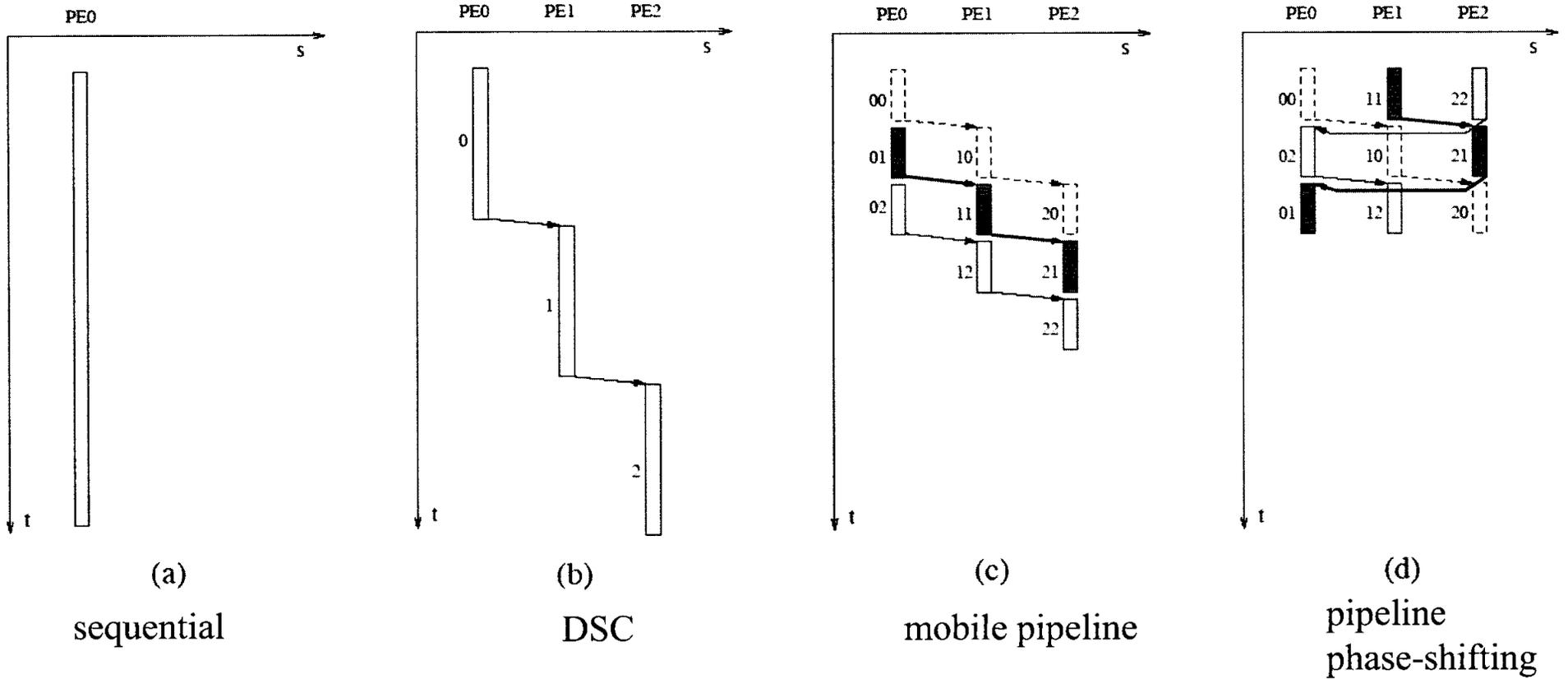


## Step 2: Sequential to DSC

- Insert hop() statements to resolve the DBlocks chosen from
  - DBlock analysis
- Following the Principle of Pivot-computes (i.e., smaller-sized data hops to meet with large-sized data for each DBlock)

## Step 3: DSC to DPC

### NavP code transformations



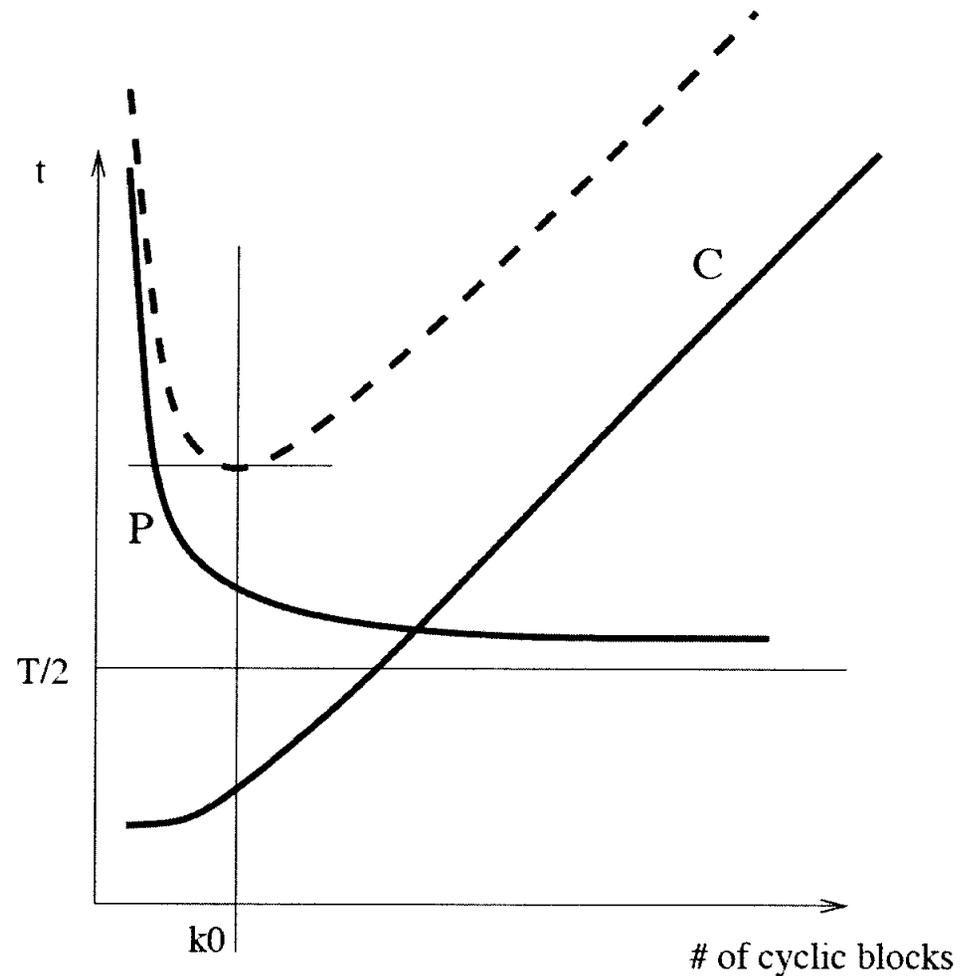
## Step 4: Loop Back

	PE1				PE2			
block	1	2	3	4	5	6	7	8
block cyclic	1	2	5	6	3	4	7	8

### ■ Refinements

- Block data distribution to block cyclic data distribution
- Refining or coarsening granularity level of computation

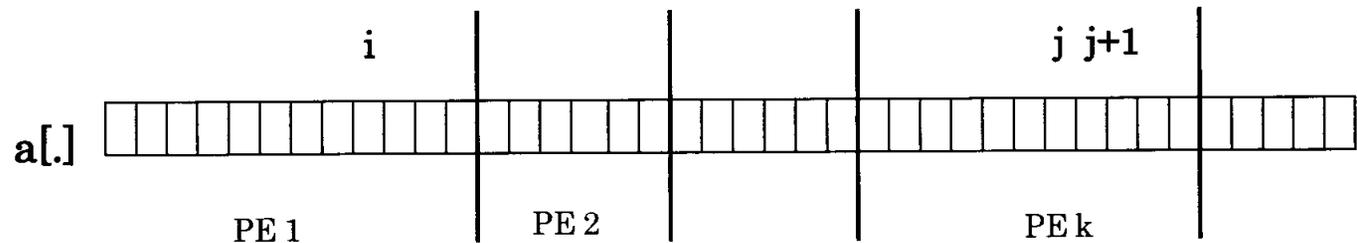
### ■ Find a balancing point



## NavP Simple Example: A Left Looking Algorithm

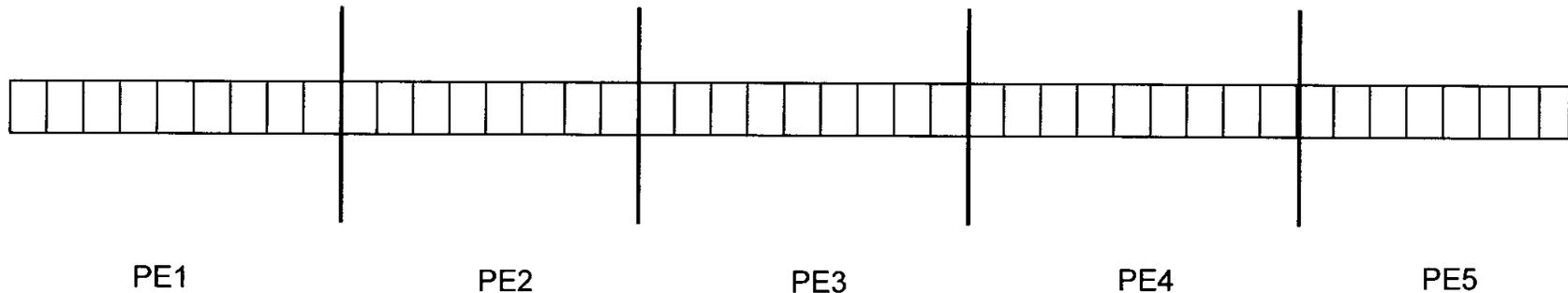
```
do j = 2 to n
  do i = 1 to j-1
    a[j] = (a[j]+a[i])*j/(j+i)
  end do /* computations not communitive, nor associative */
```

```
  a[j] = a[j]/j
end do
```



## NavP Simple Example Step 1: Data Distribution

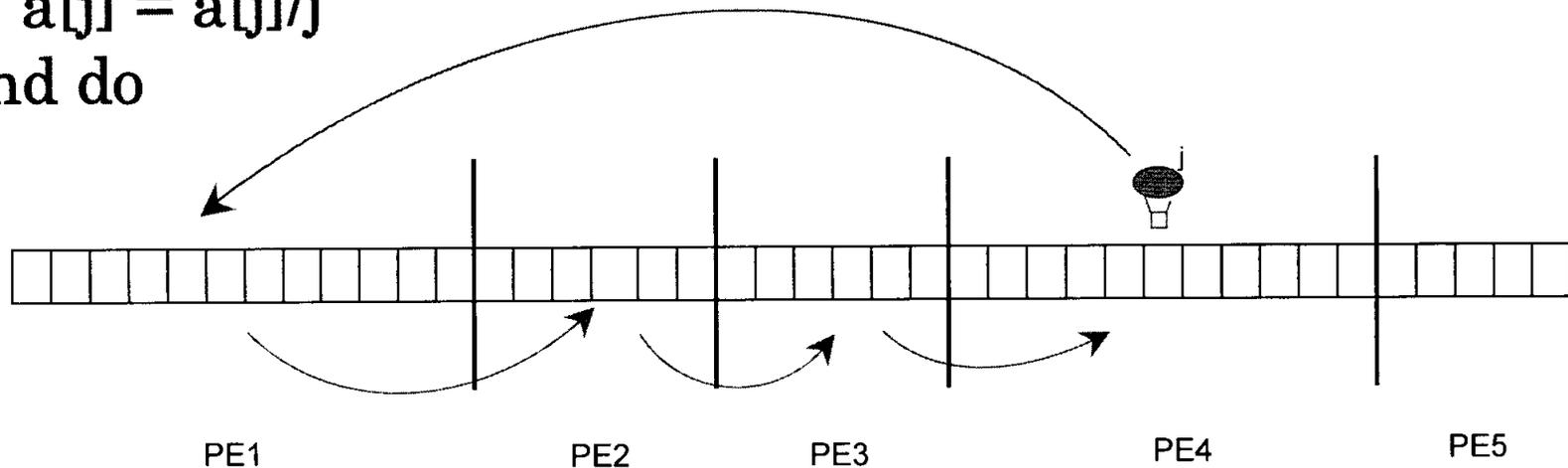
- Affinity graph:
  - **A** edges fully connect the graph
  - **C** edges connect the neighbors
- *k*-way graph partitioning
- Result: Block data distribution



## NavP Simple Example Step 2: Sequential to DSC

```
do j = 2 to n
  hop(node[j]); mx = a[j]
  do i = 1 to j-1
    hop(node[i])
    mx = (mx+a[i])*j/(j+i)
  end do
  hop(node[j]); a[j] = mx
  a[j] = a[j]/j
end do
```

- i, j, mx are “agent variables”
- a[.] is a “DSV”
- Most hops are no-ops



## NavP Simple Example Step 3: DSC to DPC

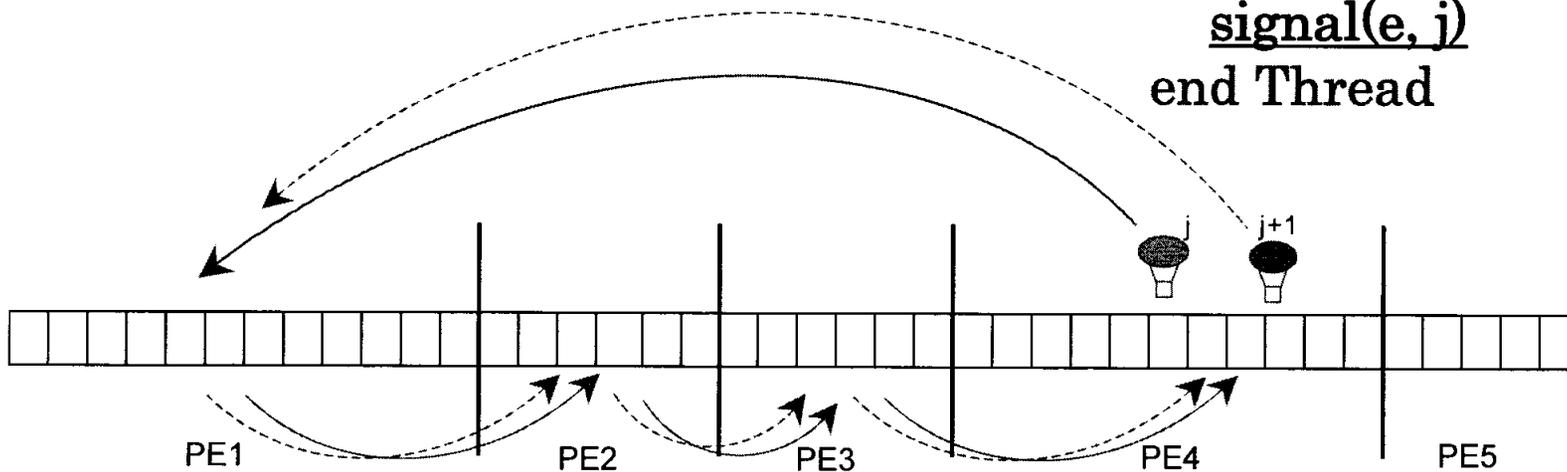
- Long DSC thread cut into shorter ones and pipelined

```

signal(e, 1)
do j = 2 to n
  inject(thrd(j))
end do
  
```

```

Thread thrd(j)
  hop(node[j]); mx = a[j]
  do i = 1 to j-1
    hop(node[i])
    wait(e, i)
    mx = (mx+a[i])*j/(j+i)
  end do
  hop(node[j]); a[j] = mx
  a[j] = a[j]/j
  signal(e, j)
end Thread
  
```



## NavP Simple Example (cont'd)

- Animation: [pipe.swf](#)
- conventional vs. mobile pipelines
  - factory work vs. farmland work
  - carpet cleaning vs. laundry
  - Chinese banquet vs. western buffet

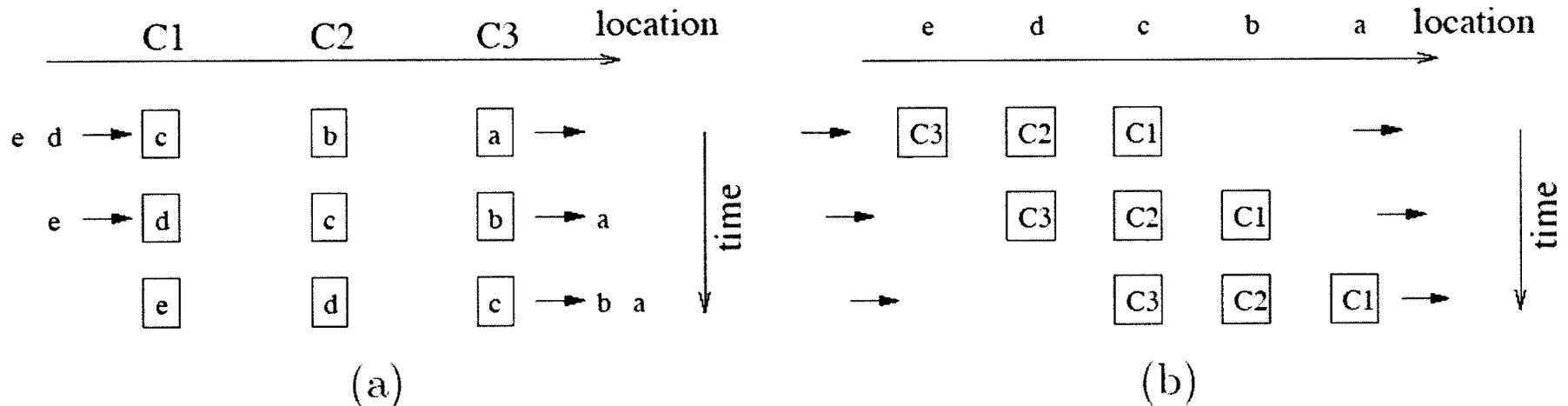


Figure 4.7. The comparison of two pipelines. (a) Conventional. (b) Mobile.

## NavP Simple Example (cont'd)

- Complexity
  - Storage on each node:  $O(N/P)$  (not  $\Theta(N)$ )
  - Total communication:  $O(N*P)$  (not  $\Theta(N^2)$ )
- Algorithmic integrity
- Other solutions
  - cache all entries ( **$\Theta(N)$  storage**)
  - recommunicate ( **$\Theta(N^2)$  communication**)
  - loop interchange (change algorithm structure)

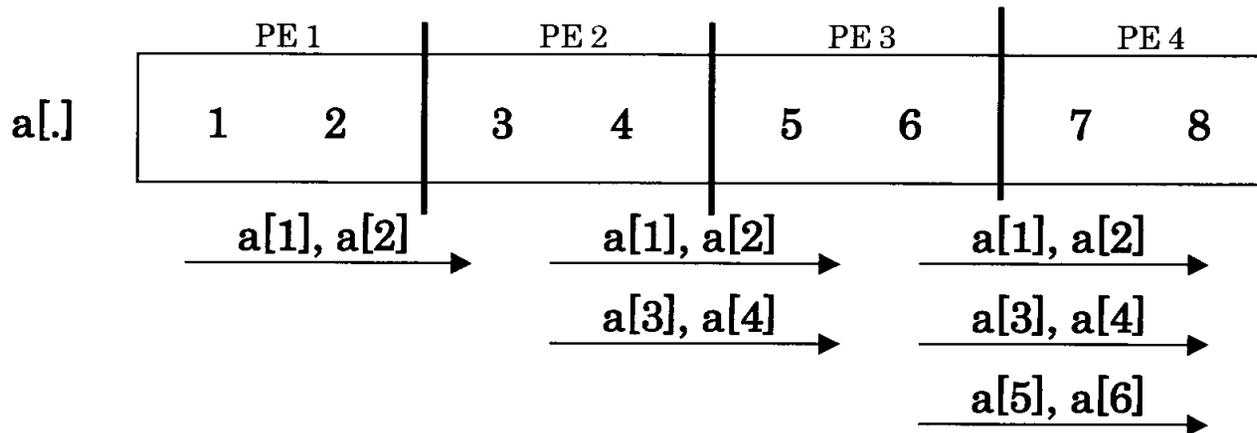
## NavP Simple Example: Pipeline Parallelism

- Using shared memory or DSM

```
post(EV(1))  
doacross j = 2 to n  
  do i = 1 to j-1  
    wait(EV(i))  
    a[j] = (a[j]+a[i])*j/(j+i)  
  end do  
  
  a[j] = a[j]/j  
  post(EV(j))  
end do
```

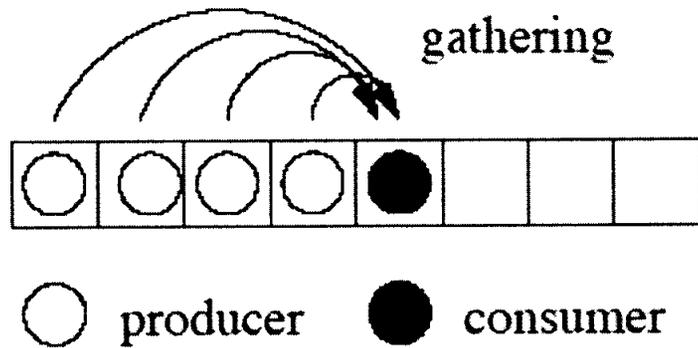
- Problems:
  - Either: not a scalable solution if some nodes are required to cache the entire array ( **$\Omega(N)$  storage**)
  - Or: not an efficient solution if a[i] entries are pulled to the nodes whenever needed ( **$\Omega(N^2)$  communication**)

## NavP Simple Example: Direct Message Passing Pipeline

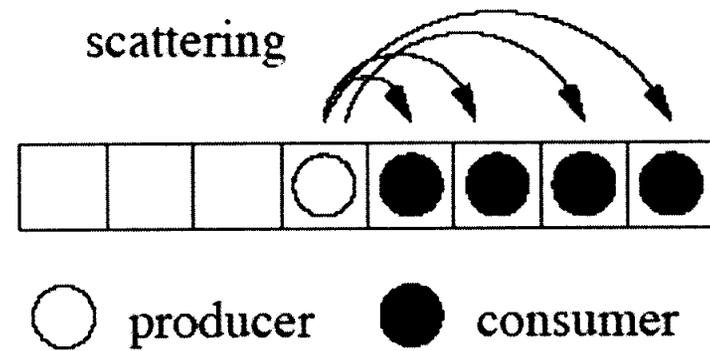


- Problem: mismatch between data flow and computation sequence
- Possible solutions:
  - Either: cache all entries ( $\Omega(N)$  storage, not scalable)
  - Or: send/recv the same entry multiple times ( $\Omega(N^2)$  communication, not efficient)
  - Or: loop interchange (not trivial and maybe dangerous)

# NavP Simple Example: Swap the Loop to Make Right Looking



(a) Left-looking pattern

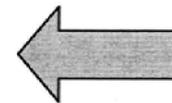


(b) Right-looking pattern

## NavP Simple Example: Swap the Loop to Make Right Looking

- Relying on loop interchange is dangerous. What if your boss stepped in and said: “Oh, for better stability/convergence, we’d like to add a new line in the algorithm ...?”

```
do j = 2 to n
  do i = 1 to j-1
    a[j] = (a[j]+a[i])*j/(j+i)
    a[i] = a[i] + 0.001*a[j]
  end do
```



New line that  
changes dependency.  
Neither left- nor  
right-looking now.

```
  a[j] = a[j]/j
end do
```

**All your previous efforts are wasted!**

- NavP handles the above situation easily (just add that line in to the NavP code)

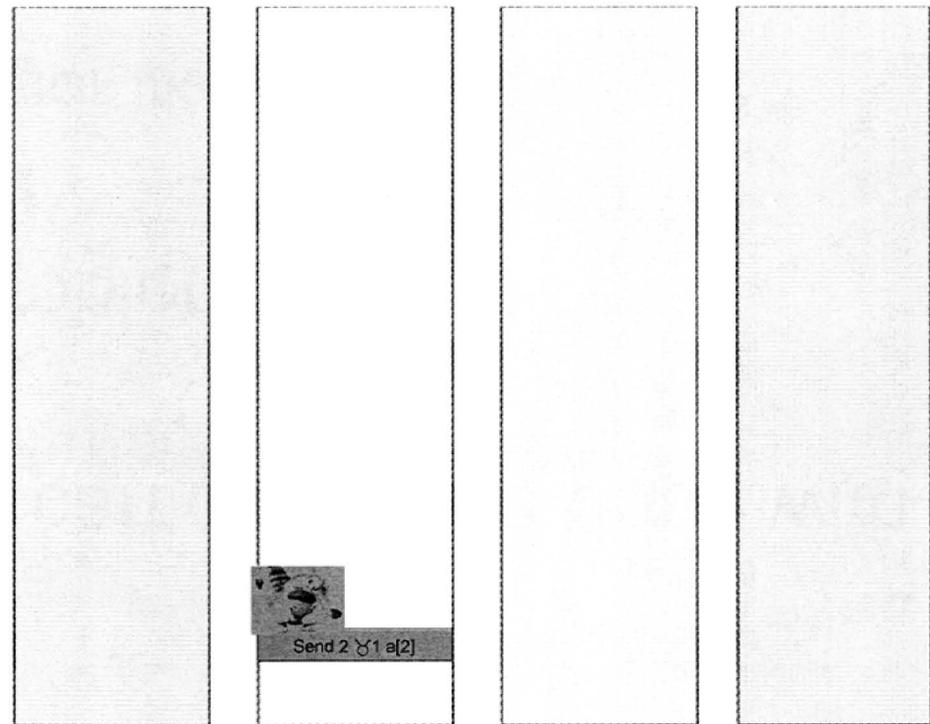
## NavP Simple Example: MP Mimicking NavP

- Anything that NavP can do can be done with MP and vice versa
- But there may be problems:
  - Loop is interchanged
  - Send()/Recv() behavior like goto/label
  - Small perturbation in the original algorithm could result in large deviation in MP code (Algorithmic Instability)

## NavP Simple Example: Card-dropping Transformation

- Transform from DSC code to MP code
- From MP to DSC, use card-collecting transformation
- Animation by Wendy Zhang

```
do j = 2 to n
  hop(node[j]); mx = a[j]
  do i = 1 to j-1
    hop(node[i])
    mx = (mx+a[i])*j/(j+i)
  end do
  hop(node[j]); a[j] = mx
  a[j] = a[j]/j
end do
```



## NavP Simple Example: MP Mimicking NavP (fine grained)

```
(1) for j = 2 to N
(2)   for i = 1 to j - 1
(3)     a[j] ← j * (a[j] + a[i]) / (j + i)
(4)   end for
(5)   a[j] ← a[j] / j
(6) end for
```

(a)

Original sequential

```
(1) i ← rank
(2) if (i ≠ 1)
(3)   Send(a[1].node[1])
(4)   Recv(x.node[i - 1])
(5)   a[1] ← x / i
(6) end if
(7) for j = i + 1 to N
(8)   if (rank == node[1])
(9)     Recv(x.node[j])
(10)  else
(11)    Recv(x.node[i - 1])
(12)  end if
(13)  x ← j * (x + a[1]) / (j + i)
(14)  Send(x.node[i + 1])
(15) end for
```

(b)

MP

- Send/rcv behave like goto, so the code structure changed
- MP program is parallel, hence is harder to debug than the DSC program

## NavP Simple Example: Small Perturbation in Original

```
do j = 2 to n by 3  
  do i = 1 to j-1 by 2  
    a[j] = (a[j]+a[i])*j/(j+i)  
  end do
```

```
  a[j] = a[j]/j  
end do
```

## means large deviation in MP

```
(1) i ← rank
(2) if ((i - 2)%3 == 0)
(3)   Send(a[1].node[1])
(4)   Recv(x,node[left_i(i)])
(5)   a[1] ← x/i
(6) end if

(7) if ((i - 1)%2 == 0)
(8)   for j = right_j(i) to N by 3
(9)     if (rank == node[1])
(10)      Recv(x,node[j])
(11)     else
(12)      Recv(x,node[left_i(i)])
(13)     end if

(14)     x ← j * (x + a[1]) / (j + i)
(15)     Send(x,node[min(right_i(i), j)])
(16)   end if
(17) end for
```

## Algorithmic Instability!

```
// the 1st i value to this PE's left
(18) int Function left_i(int i)
(19)   i1 ← i - 1
(20)   while ((i1 - 1)%2 != 0)
(21)     i1 ← i1 - 1
(22)   end while

(23)   return i1
(24) end Function

// the 1st i value to this PE's right
(25) int Function right_i(int i)
(26)   i1 ← i + 1
(27)   while ((i1 - 1)%2 != 0)
(28)     i1 ← i1 + 1
(29)   end while

(30)   return i1
(31) end Function

// the 1st j value to this PE's right
(32) int Function right_j(int i)
(33)   i1 ← i + 1
(34)   while ((i1 - 2)%3 != 0)
(35)     i1 ← i1 + 1
(36)   end while

(37)   return i1
(38) end Function
```

## NavP Simple Example: MP Mimicking NavP (coarse grained)

```

// send to the node that hosts a[1]
(1)  recv_cnt ← 0
(2)  for l1 = 1 to local_cnt
(3)    l2 ← oa[l1]; j ← g[l2]
(4)    recv_cnt ← recv_cnt + N - j
(5)    if (j ≠ 1)
(6)      recv_cnt ← recv_cnt + 1
(7)      if (rank ≠ node_map[1])
(8)        Send(a[l1], j, node_map[1])
(9)      end if
(10)   else
(11)     recv_cnt ← recv_cnt - 1
(12)     enqueue(Q[rank], a[l1], j, 1)
(13)   end if
(14) end for

// compute recv_cnt
(15) l2 ← oa[1]
(16) g2 ← g[l2]
(17) for l1 = 2 to local_cnt
(18)  l3 ← oa[l1]; g3 ← g[l3]
(19)  if (g2 + 1 == g3)
(20)    recv_cnt ← recv_cnt - (N - g3 + 1)
(21)  end if
(22)  g2 ← g3
(23) end for

// the node that hosts the 1st entry
(24) if (rank == node_map[1])
(25)  for l1 = 2 to N
// check if any queue has this entry
(26)    l1_flg ← 0
(27)    for l2 = 1 to num_nodes
(28)      if (Q[l2] not empty)
(29)        if (l1 found)
(30)          dequeue(Q[l2], x, j, 1)
(31)          Call request_proc(1, j, x)
(32)          l1_flg ← 1
(33)          break
(34)        end if
(35)      end if
(36)    end for

```

```

// not in the queues, so receive
(37)  if (l1_flg == 0)
(38)    while (recv_cnt > 0)
(39)      recv(x, j, ANY_SRC)
(40)      src_rank ← the sender's rank
(41)      recv_cnt ← recv_cnt - 1
(42)      i ← 1
// if received not the one, enqueue it
(43)      if (l1 ≠ j)
(44)        enqueue(Q[src_rank], x, j, 1)
(45)      else
(46)        break
(47)      end if
(48)    end while
(49)    Call request_proc(1, j, x)
(50)  end if
(51) end for
(52) end if

// receive until all msgs received
(53) while (recv_cnt > 0)
(54)  recv(x, j, 1, ANY_SRC)
(55)  recv_cnt ← recv_cnt - 1
(56)  Call request_proc(1, j, x)
(57) end while

(58) Procedure request_proc(1, j, x)
(59)  for l1 = 1 to j - 1
(60)    if (rank == node_map[l1])
(61)      x ← j * (x + a[1[l1]]) / (j + 1)
(62)    else
(63)      break
(59)  end if
(60) end for

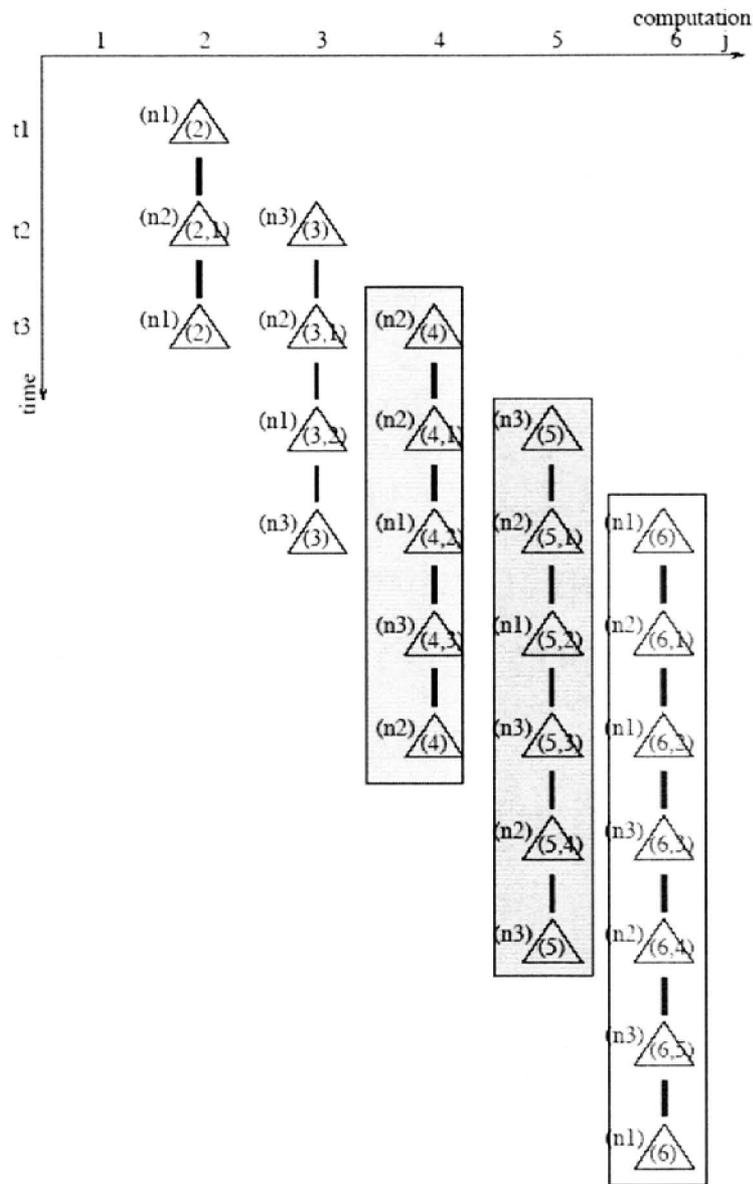
(61) if (l1 == j And rank == node_map[j])
(62)  a[1[j]] ← x / j
(63) else
(64)  send(x, j, l1, node_map[l1])
(65) end if
(66) end Procedure

```

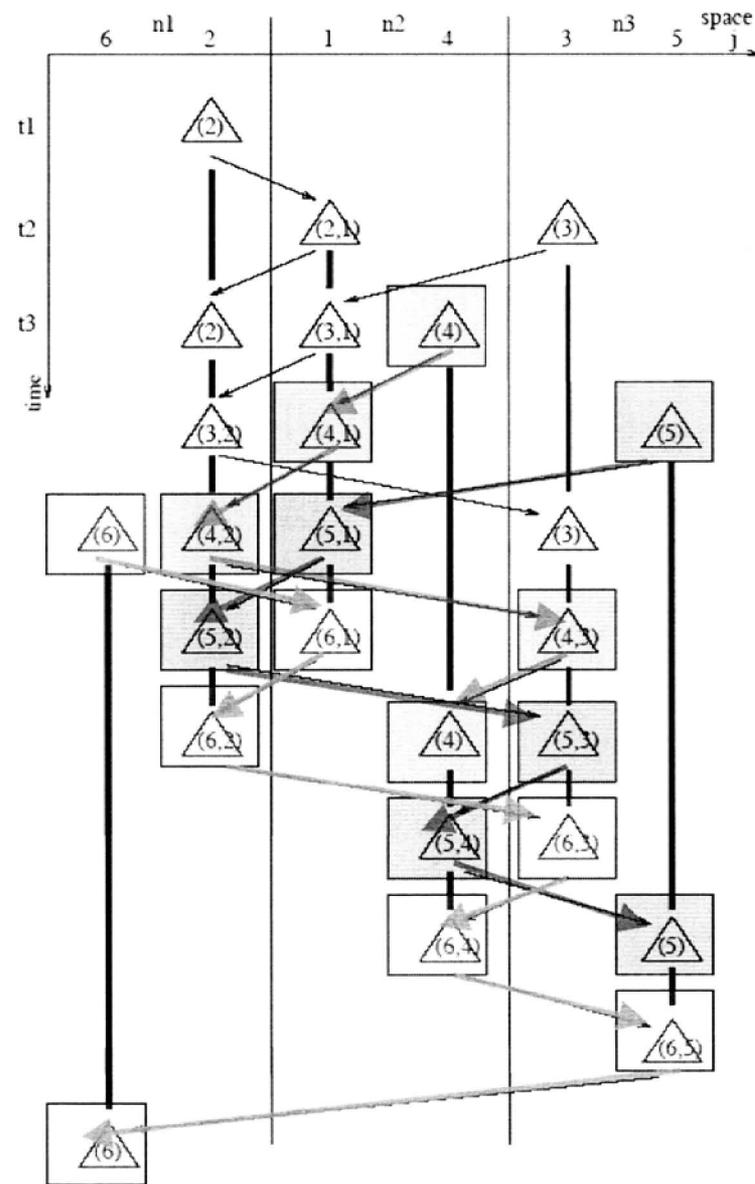
## Difficulties with MP Implementation

- **Adapting to arbitrary data distribution**
  - Irregular communication pattern
  - Partial sorting and queuing needed
- **Code restructuring**
  - Loops broken
  - Repeated core code lines
  - Large amount of auxiliary code
- **Termination complicated**
  - Anticipate number of messages a node will receive
- **Self-sending using shared memory**
  - Use queues (sending to a node itself is expensive with MPI)
- **Deadlock**
  - Balance receiving, processing, and sending (need `MPI_THREAD_MULTIPLE`)
  - Or learn the tricks of posting `recv`'s at the right place

# Regular vs. Irregular Communication Patterns



NavP



MP

## NavP Simple Example:MP or NavP?

- MP is excellent for client-server applications (computation described at stationary locations)
- Our suggestion: MP is not necessarily a bad way to implement, but NavP is a more structured way to design and describe general purpose distributed parallel algorithms
  - NavP is the “source code”
  - MP is the “target code”
    - OK to have “jump”s in assembly code
    - Likewise OK to have send/recv in MPI -- the “assembly code” for distributed computing
  - The compiler from NavP to MP exists
  - The compiler from sequential to NavP is to be built

## NavP Simple Example Step 4: Loop Back

- Performance considerations
  - Block algorithm (coarser level computations)
    - Can be automated using loop tiling techniques
  - Block cyclic data distribution (more parallelism)
    - Can be automated (NavP code works for arbitrary data distribution)
  - FIFO pipeline (less signal/wait events)

## NavP Simple Example: Start from Block Algorithm

```
(1) for J = 1 to num_blocks
(1.1) hop(node_map[J]); load blk J to x[]
(2)   for I = 1 to J - 1
(2.1)   hop(node_map[I])
(3)     Call F(I, J, a, x)
(4)   end for
(4.1) hop(node_map[J])
(5)   Call F(J, J, x, x); unload x[] to blk J
(6) end for
```

(a)

```
(1) Procedure F(I, J, a, x)
(2)   for j = start(J) to end(J)
(3)     for i = start(I) to min(end(I), j - 1)
(4)       x[l[j]] ← j * (x[l[j]] + a[l[i]]) / (j + i)
(5)     end for
(6)   if (I = J) a[l[j]] ← a[l[j]] / j
(7) end for
(8) end Procedure
```

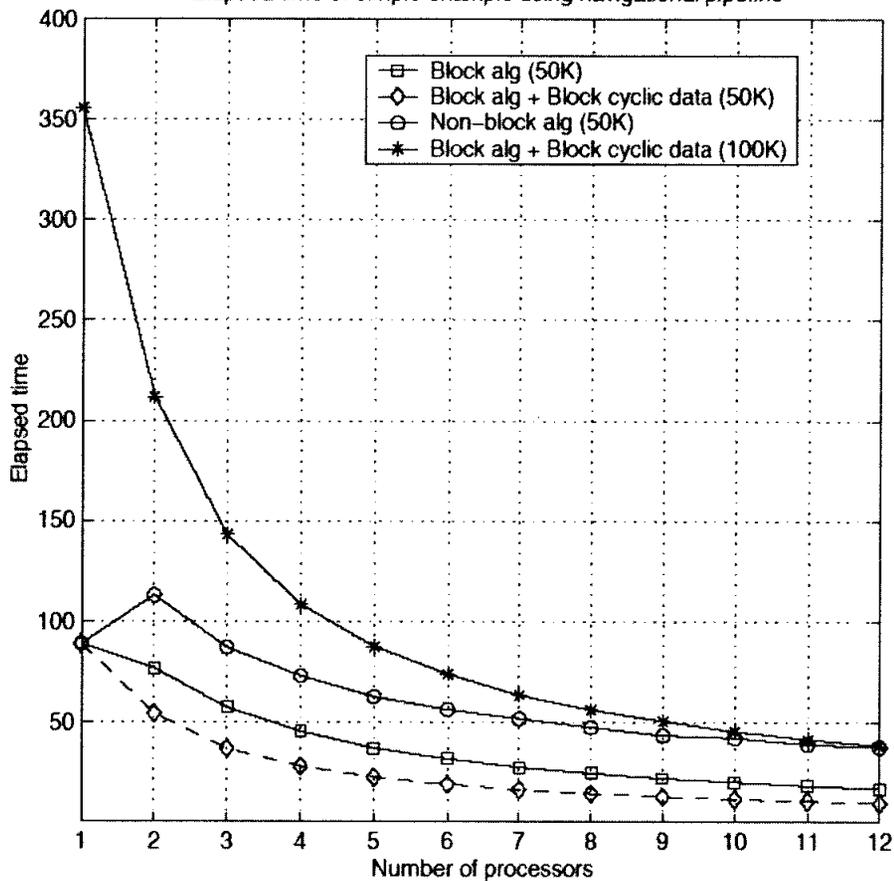
(b)

Fig. 6. Block pseudocode for the simple algorithm. (a) DSC; (b) The function F()

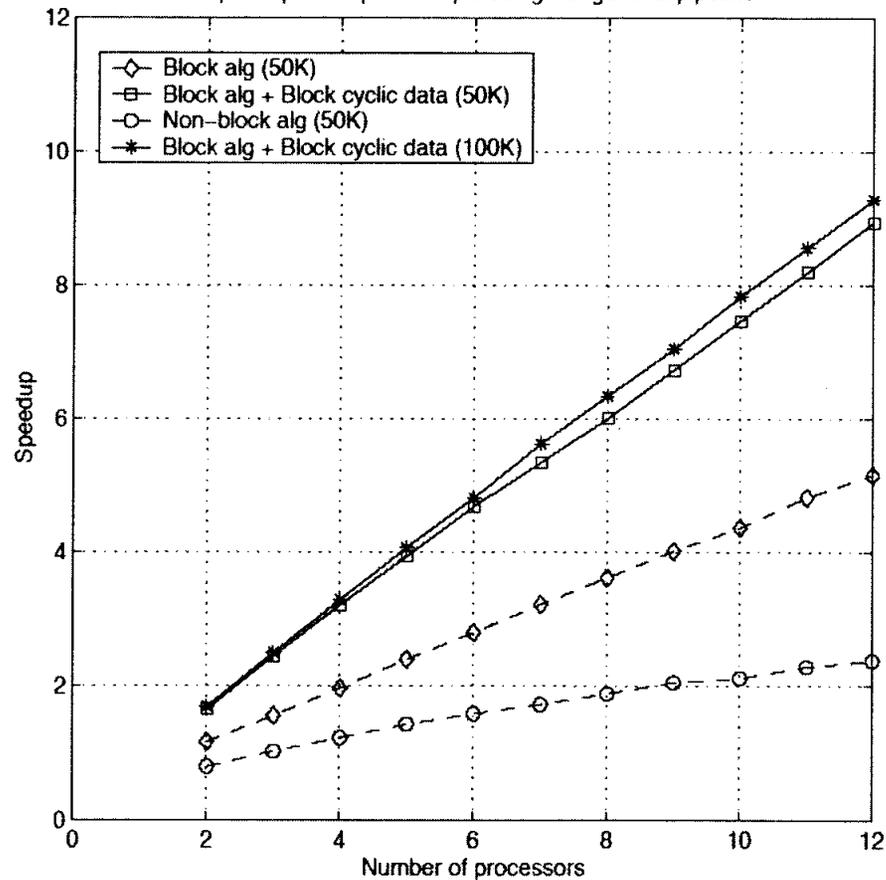
- A block algorithm is a good starting point
  - cache performance for sequential code
  - +coarse granularity for distributed code
- Can use manual programming
- Or use a loop tiling tool

# NavP Simple Example: Performance

Elapsed time of simple example using navigational pipeline



Speedup of simple example using navigational pipeline



## NavP Simple Example: Block Cyclic Data Distribution

	PE1				PE2			
block	1	2	3	4	5	6	7	8
block cyclic	1	2	5	6	3	4	7	8

- Block cyclic data distribution is a classical approach to load balancing

## Summary: The Methodology of NavP

- Presented the methodology and steps of NavP
- Used a simple left-looking example to exercise the NavP steps
- Introduced the card-dropping transformation
- Contrasted the NavP implementation to other possible implementations (MP or DSM)

## Outline

1. Introduction
2. The NavP view vs. the SPMD view
3. Distributed sequential computing (DSC)
4. The methodology of NavP
5. **Case Studies**
6. The enabling technology of NavP
7. Comparisons, conclusions, and future work

## Section 5: Case Studies

### Case selection rationale

- Solving linear systems accounts for the majority of CPU time in numerical computation
- Direct solvers are harder to parallelize than iterative solvers

### Real-world examples

- **Crout factorization:** A “left-looking” matrix factoring algorithm
- **Cholesky factorization:** A “right-looking” matrix factoring algorithm
- **Matrix multiplication:** A lot of parallelism

### Performance

- NavP implementations of the algorithms perform well
- NavP Cholesky code performs as fast as ScaLAPACK
- NavP matrix multiplication performs faster than ScaLAPACK

## Case Study 1: Crout Factorization

- A left-looking algorithm to factorize symmetric matrices

$$\mathbf{A} = \mathbf{U}^T \mathbf{D} \mathbf{U}$$

```
(1) For j = 1..N
(2)   For i = 1..j - 1
(3)      $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$ 
(4)   End For

(5)   For i = 1..j - 1
(6)      $T \leftarrow K_{ij}$ 
(7)      $K_{ij} \leftarrow \frac{T}{K_{ii}}$ 
(8)      $K_{jj} \leftarrow K_{jj} - TK_{ij}$ 
(9)   End For
(10) End For
```

Sequential

```
(1) For j = 1..N
(1.1) hop(node[j]); load(column j)
(2)   For i = 1..j - 1
(2.1)   hop(node[i]); load( $K_{ii}$ )
(3)      $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$ 
(4)   End For

(4.1) hop(node[j]); unload(column j,  $K_{ii}[]$ )

(5)   For i = 1..j - 1
(6)      $T \leftarrow K_{ij}$ 
(7)      $K_{ij} \leftarrow \frac{T}{K_{ii}}$ 
(8)      $K_{jj} \leftarrow K_{jj} - TK_{ij}$ 
(9)   End For
(10) End For
```

DSC

## Case Study 1: Crout Factorization (cont'd)

```

(1)  For j = 1..N
(2)    call col_proc(j)
(3)  End For

(4)  Procedure col_proc(int j)
(5)    For i = 1..j - 1
(6)       $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$ 
(7)    End For

(8)    For i = 1..j - 1
(9)       $T \leftarrow K_{ij}$ 
(10)      $K_{ij} \leftarrow \frac{T}{K_{ii}}$ 
(11)      $K_{jj} \leftarrow K_{jj} - TK_{ij}$ 
(12)    End For
(13) End Procedure

```

```

(1)  For j = 1..N
(2)    inject(col_proc(j))
(3)  End For

(4)  Agent col_proc(int j)
(4.1) hop(node[j]); load(column j)
(5)    For i = 1..j - 1
(5.1)   hop(node[i]); load( $K_{ii}$ )
(5.2)   If ( $j > 1$  And  $i = 1$ ) waitEvent(evt, j - 1)
(6)      $K_{ij} \leftarrow K_{ij} - \sum_{l=1}^{i-1} K_{li}K_{lj}$ 
(6.1)   If ( $i = 1$ ) signalEvent(evt, j)
(7)    End For

(7.1) hop(node[j]); unload(column j,  $K_{ii}[]$ )

(8)    For i = 1..j - 1
(9)       $T \leftarrow K_{ij}$ 
(10)      $K_{ij} \leftarrow \frac{T}{K_{ii}}$ 
(11)      $K_{jj} \leftarrow K_{jj} - TK_{ij}$ 
(12)    End For
(13) End Agent

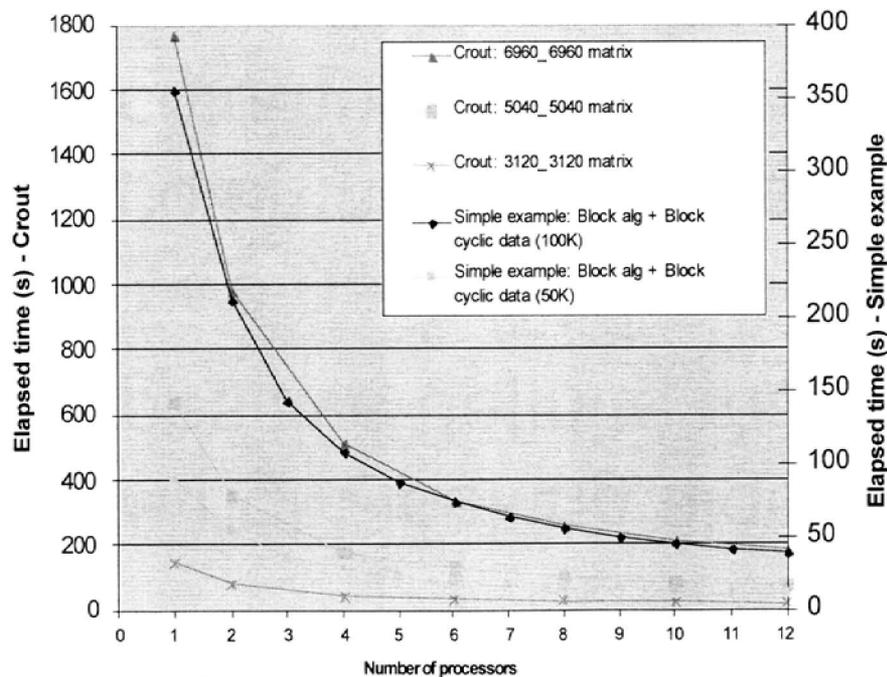
```

## Case Study 1: Crout Factorization (cont'd)

- MP implementation is left as an open problem
- NavP code performance

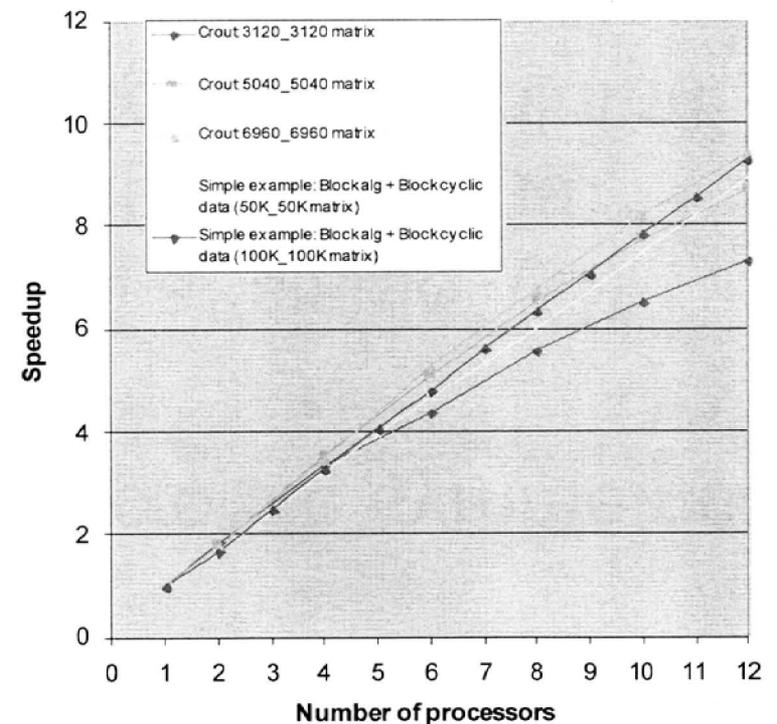
### Elapsed time of simple example and Crout factorization using navigational pipeline

- Sun Ultra60s 450MHz 256MB main memory 1GB virtual memory, 100Mbps Ethernet



### Speedup of simple example and Crout factorization using navigational pipeline

- Sun Ultra60s 450MHz 256MB main memory 1GB virtual memory, 100Mbps Ethernet



## Case Study 2: Cholesky Factorization

$$\mathbf{A} = \mathbf{G} \mathbf{G}^T$$

- A right-looking algorithm to factorize symmetric positive definite matrices
- Two nested loops, “scaling” to a “single” (G) column in outer loop, “updating” to “all” the columns in inner loop using the new G column
- In DSC code, a “scaling” on one node (different for different loops), followed by a “updating” on all the nodes. These are in loops.
- In DPC code, parallel “updatings” interspersed with sequential “scalings.”



## Case Study 2: Cholesky Factorization (cont'd)

```
(1) for k = 1 to n
(2)   if (rank == 1)
(3)     vloc(k : n) = A(k : n, k)
(4)     vloc(k : n) / = √vloc(k)
(5)     A(k : n, k) = vloc(k : n)
(6)   end if
(7)   barrier

(8)   call updating(rank, k, n)

(9)   barrier

(10) end for

(11) updating(int rank, int k, int n)
(12)   vloc(k + 1 : n) = A(k + 1 : n, k)

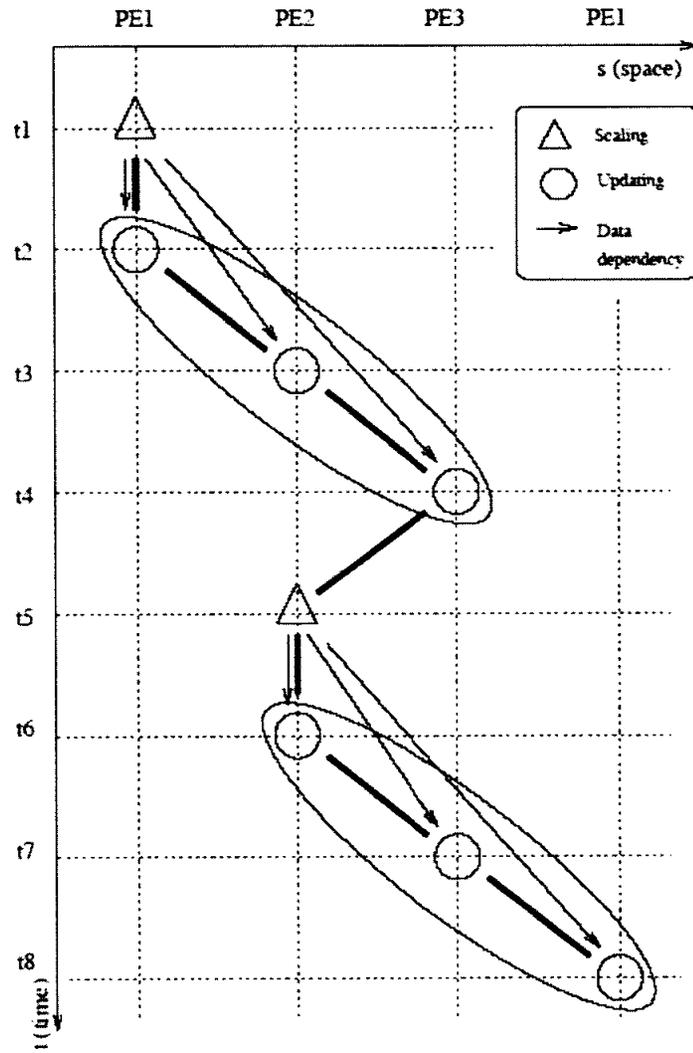
(13)   for j = k + rank to n by p
(14)     wloc(j : n) = A(j : n, j)
(15)     wloc(j : n) - = vloc(j)vloc(j : n)
(16)     A(j : n, j) = wloc(j : n)
(17)   end for

(18) end
```

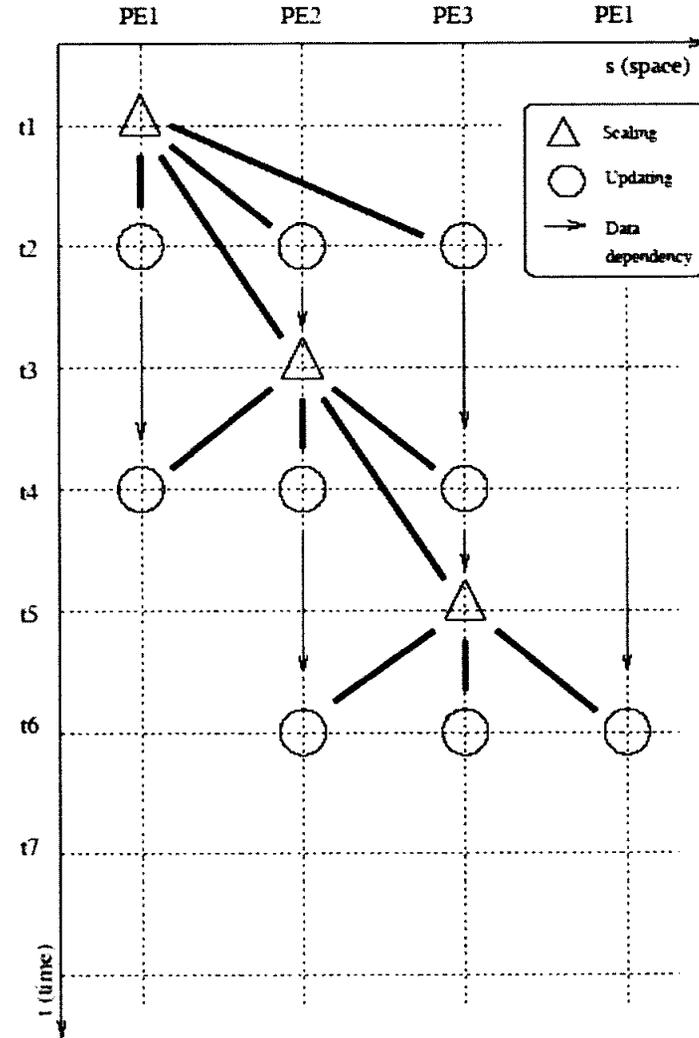
```
(1) for k = 1 to n
(2)
(3)
(4)   A(k : n, col(k)) / = √A(k, col(k))
(5)
(6)
(7)
(7.1) for rank = 1 to p
(8)   inject(updating(rank, k, n))
(8.1) end for
(9)
(9.1) hop(node_map(k + 1))
(9.2) waitEvent(Evt, k + 1)
(10) end for

(11) updating(int rank, int k, int n)
(12)   vloc(k + 1 : n) = A(k + 1 : n, col(k))
(12.1) hop(node_map(k + rank))
(12.2) waitEvent(Evt, k)
(13)   for j = k + rank : p : n
(14)
(15)     A(j : n, col(j)) - = vloc(j)vloc(j : n)
(16)
(17)   end for
(17.1) signalEvent(Evt, k + 1)
(18) end
```

# Case Study 2: Cholesky Factorization (cont'd)



DSC



DPC

## Cholesky Factorization: MP Code

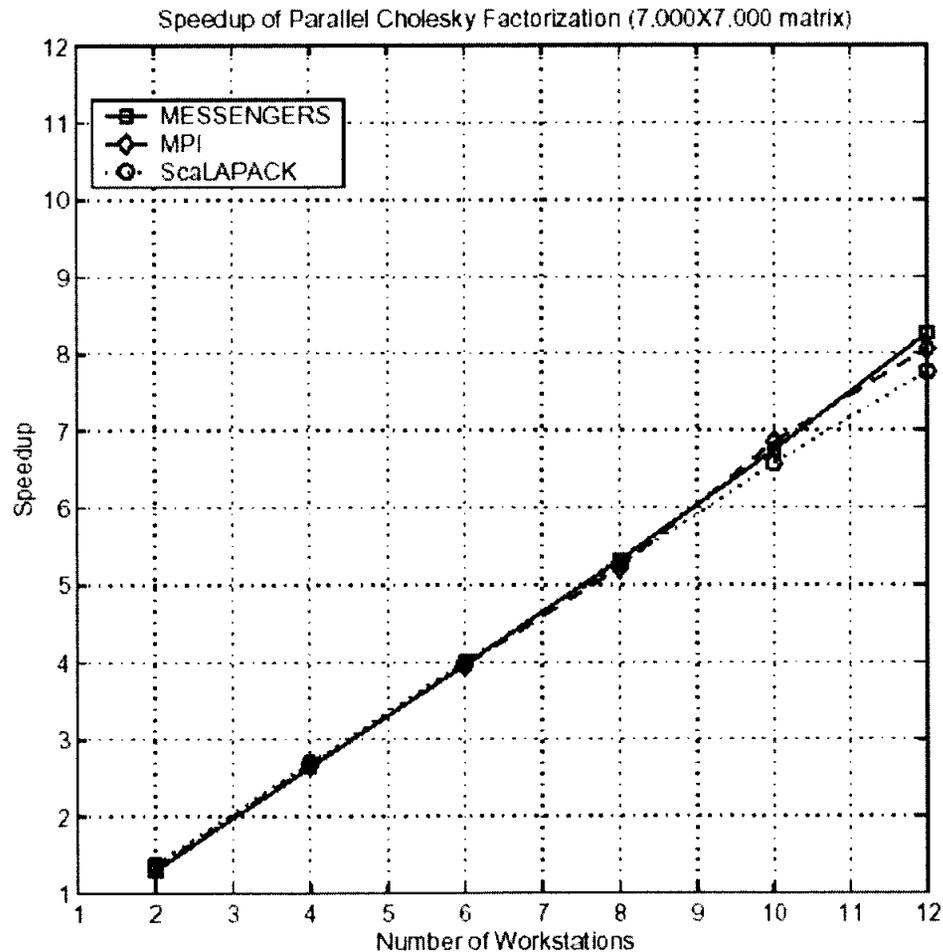
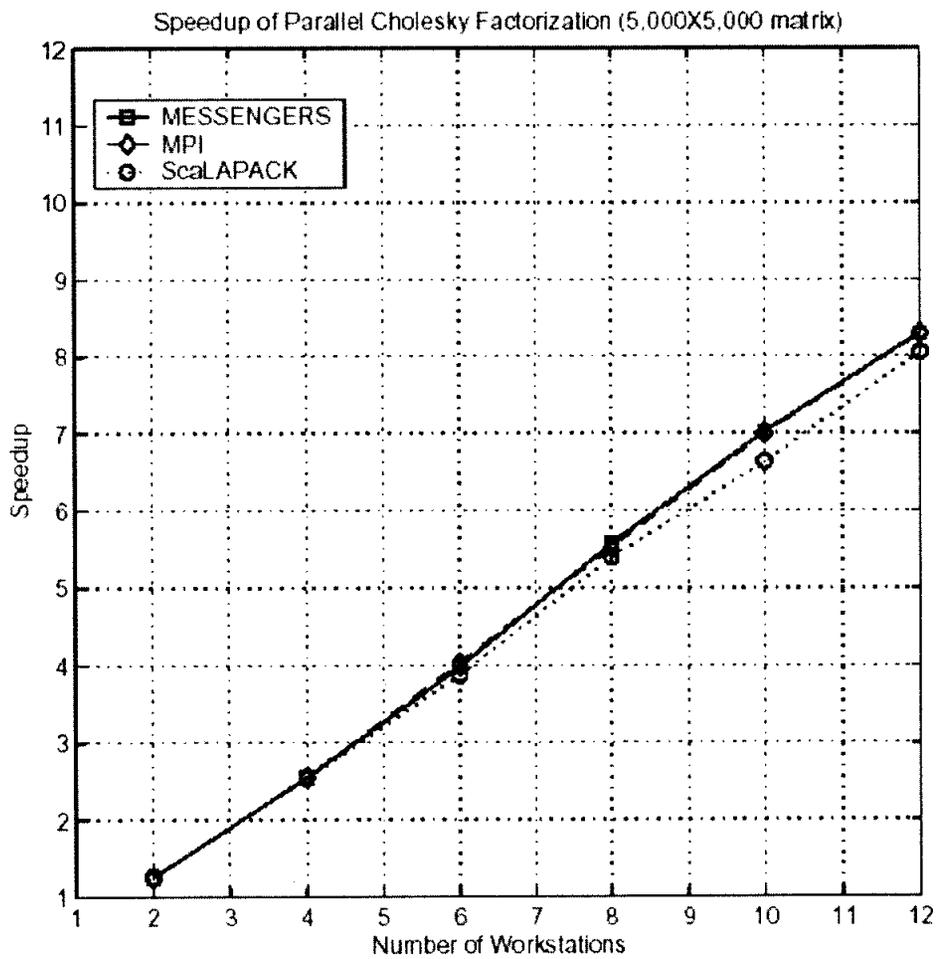
```
(1)  $k = 1; q = 1; col = \mu : p : n; L = length(col)$ 
(2) while  $q \leq L$ 
(3) if  $k == col(q)$ 
(4)  $A_{loc}(k : n, q) /= \sqrt{A_{loc}(k, q)}$ 
(5) if  $k < n$ 
(6)  $Send(A_{loc}(k : n, q), right)$ 
(7) end if
(8)  $k = k + 1$ 
(9) for  $i = q + 1 : L$ 
(10)  $r = col(i)$ 
(11)  $A_{loc}(r : n, i) -= A_{loc}(r, q)A_{loc}(r : n, q)$ 
(12) end for
(13)  $q = q + 1$ 
(14) else
(15)  $Recv(g_{loc}(k : n), left)$ 
(16)  $\alpha = proc$  which sent  $k^{th}$   $G$  col
(17)  $\beta = index$  of right's final col
(18) if  $right \neq \alpha$  and  $k < \beta$ 
(19)  $Send(g_{loc}(k : n), right)$ 
(20) end if
(21) for  $i = q : L$ 
(22)  $r = col(i)$ 
(23)  $A_{loc}(r : n, i) -= g_{loc}(r)g_{loc}(r : n)$ 
(24) end for
(25)  $k = k + 1$ 
(26) end if
(27) end while
```

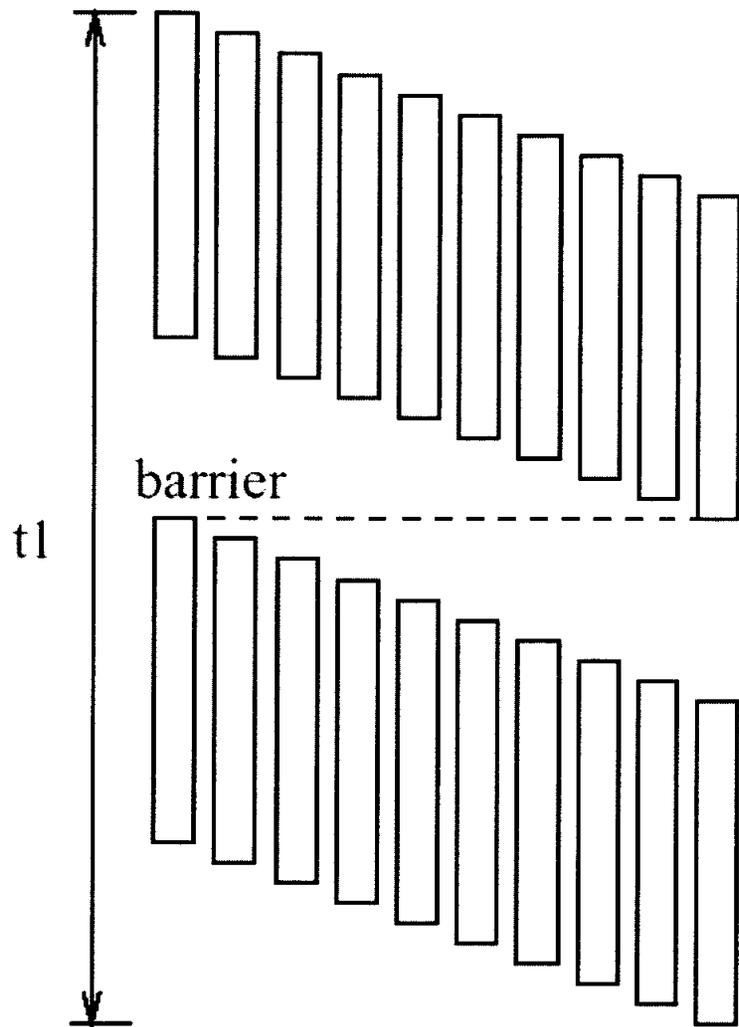
**MP**

- 1). Both loops broken into smaller ones over locally owned columns;
- 2). if()/else if() for different nodes;
- 3). Local column index start from 1;
- 4). Termination condition complicated.
- 5). Repeated core code

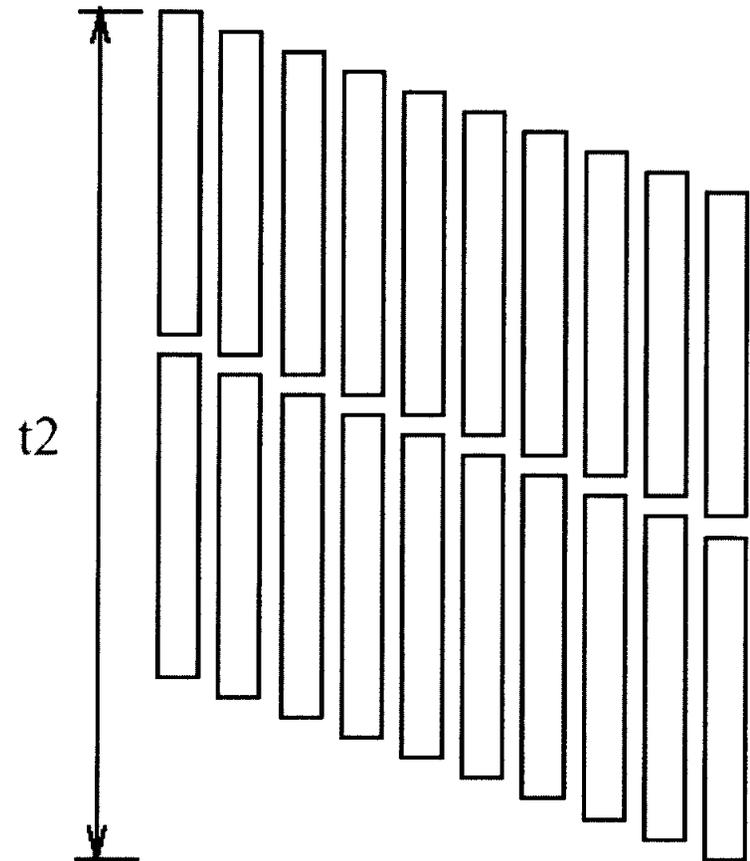
## Case Study 2: Cholesky Factorization (cont'd)

The NavP code performs as well as the MP code or ScaLAPACK.





**In DSM, global sync with *barriers***



**In NavP, local sync with events**

**NavP uses SV programming as DSM does, but it does not use *barriers***

## Case Study 3: Matrix Multiplication

### ■ Sequential

```
(1) for i = 0 to p - 1
(2)   for j = 0 to p - 1
(3)     Cij = Ai Bj
(4)   end for
(5) end for
```

### ■ Abundant parallelism. But doall's work for distributed memory?

- Cache A and B everywhere; or
- Contention

```
(1) doall i=0,N-1
(2)   doall j=0,N-1
(3)     C(i,j) = 0.0
(4)     do k=0,N-1
(5)       C(i,j) += A(i,k) * B(k,j)
(6)     end do
(7)   end doall
(8) end doall
```

## Case Study 3: Matrix Multiplication (cont'd)

```
(1)  do k=0,N-2
(2)    doall node(i,j) where 0<=i,j<=N-1
(3)      if i>k then
(4)        A ← east(A)
(5)      end if
(6)      if j>k then
(7)        B ← south(B)
(8)      end if
(9)    end do
(10) end do

(11) doall node(i,j) where 0<=i,j<=N-1
(12)  C = A * B
(13) end do
(14) do k=0,N-2
(15)  doall node(i,j) where 0<=i,j<=N-1
(16)    A ← east(A)
(17)    B ← south(B)
(18)    C += A * B
(19)  end do
(20) end do
```

- Gentleman's Algorithm
  - How did this guy get here?

## Case Study 3: Matrix Multiplication (cont'd)

```
(1) hop(node(0))
(2) inject(RowCarrier)

(1) RowCarrier
(2)   do mi=0,N-1
(3)     do mj=0,N-1
(4)       hop(node(mj))
(5)       if(mj=0) mA(*) = A(mi,*)
(6)       t = 0.0
(7)       do k=0,N-1
(8)         t += mA(k) * B(k)
(9)       end do
(10)      C(mi) = t
(11)    end do
(12)  end do
(13) end
```

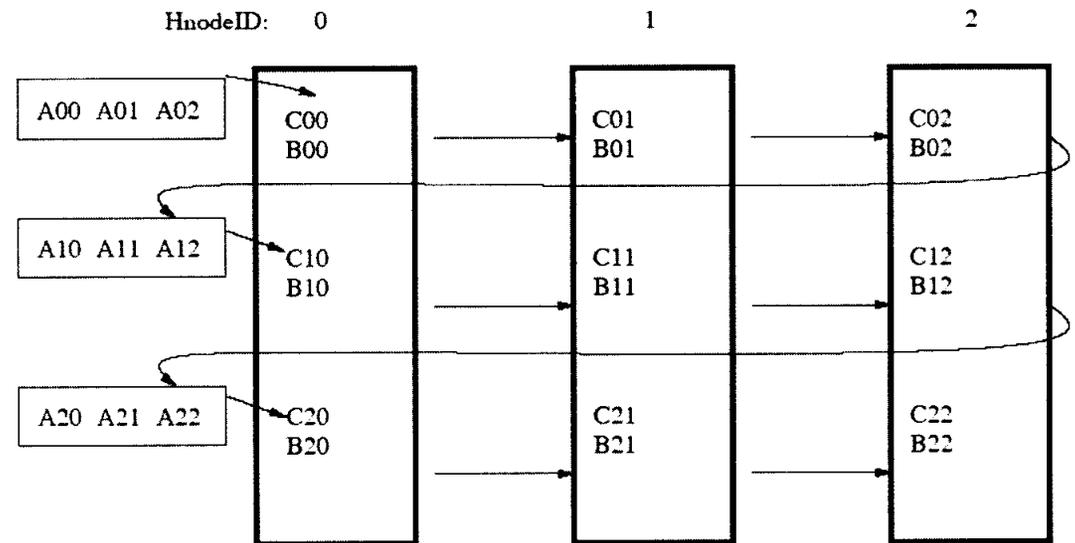


Figure 5.16. DSC.





# Case Study 3: Matrix Multiplication (cont'd)

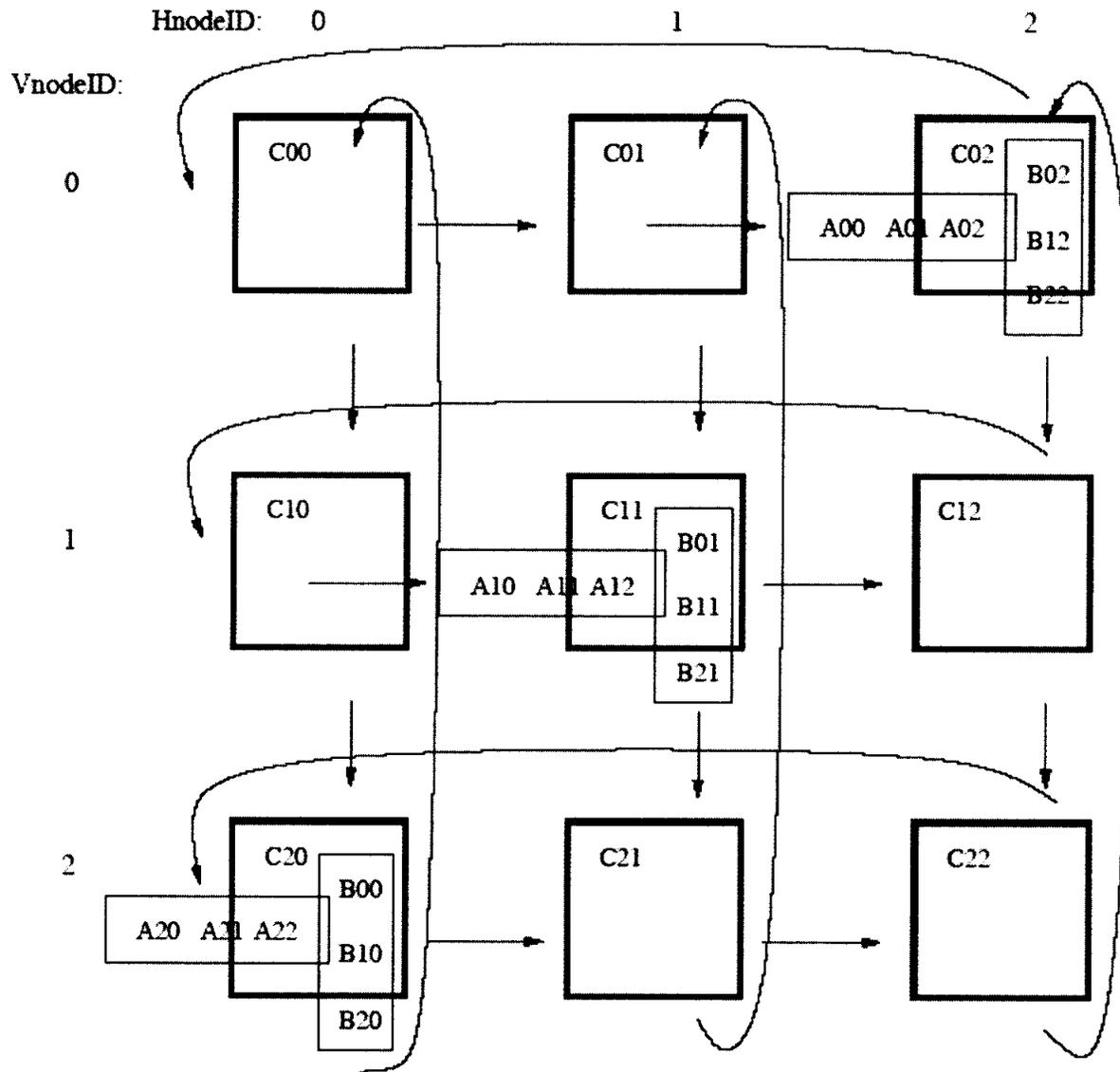


Figure 5.22. DSC in the second dimension.

## Case Study 3: Matrix Multiplication (cont'd)

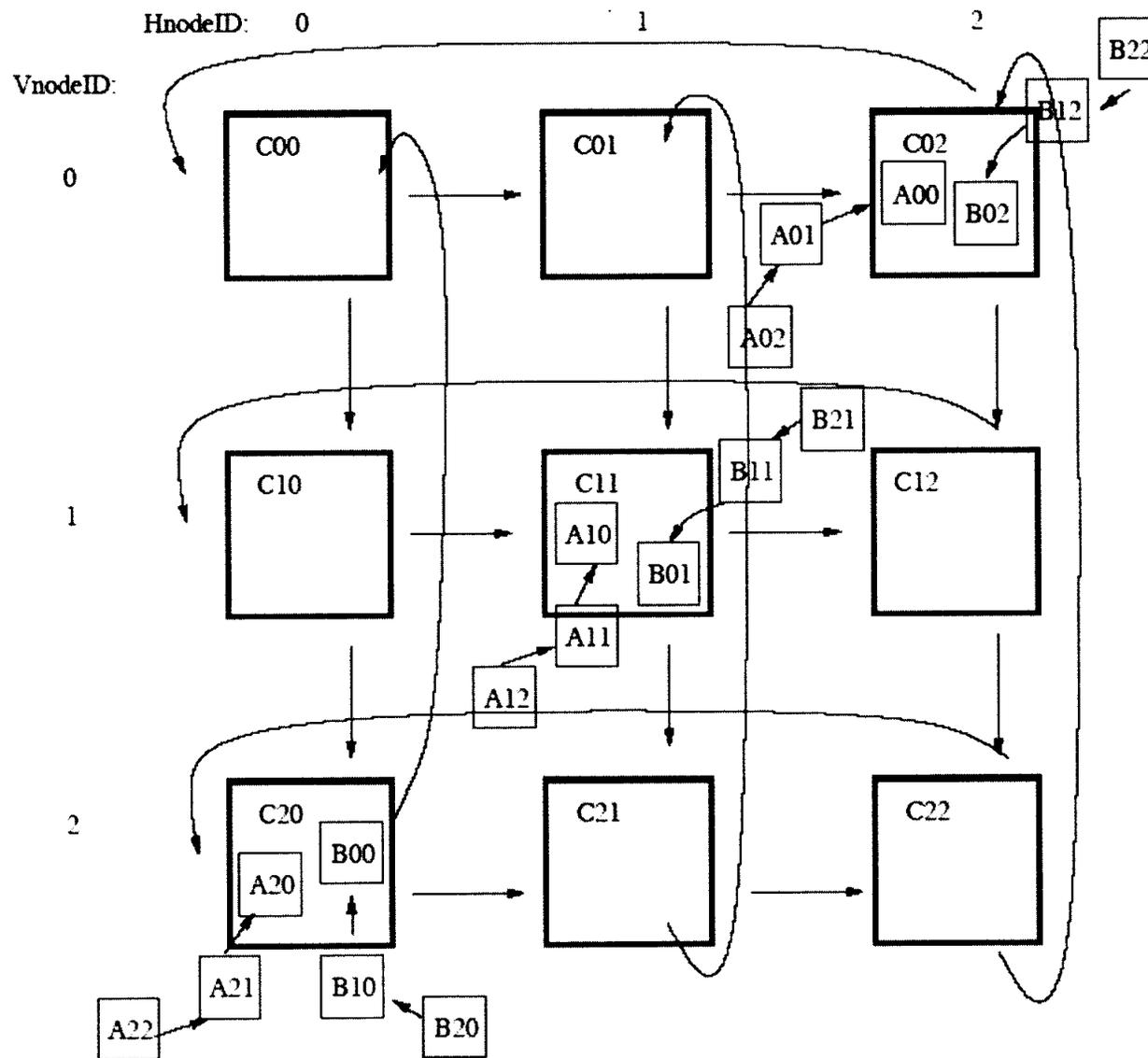


Figure 5.24. DSC pipelining in both dimensions.

## Case Study 3: Matrix Multiplication (cont'd)

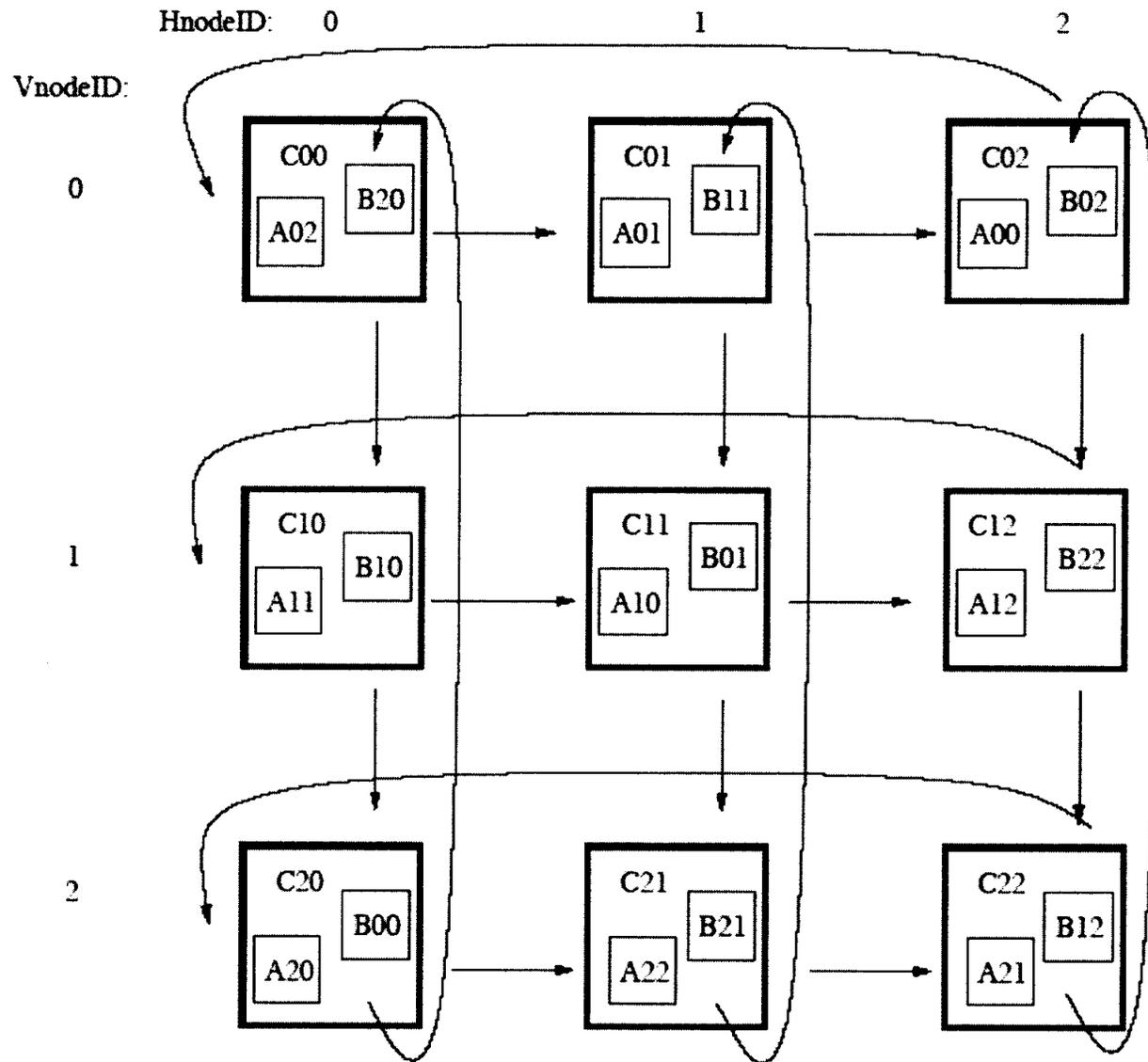


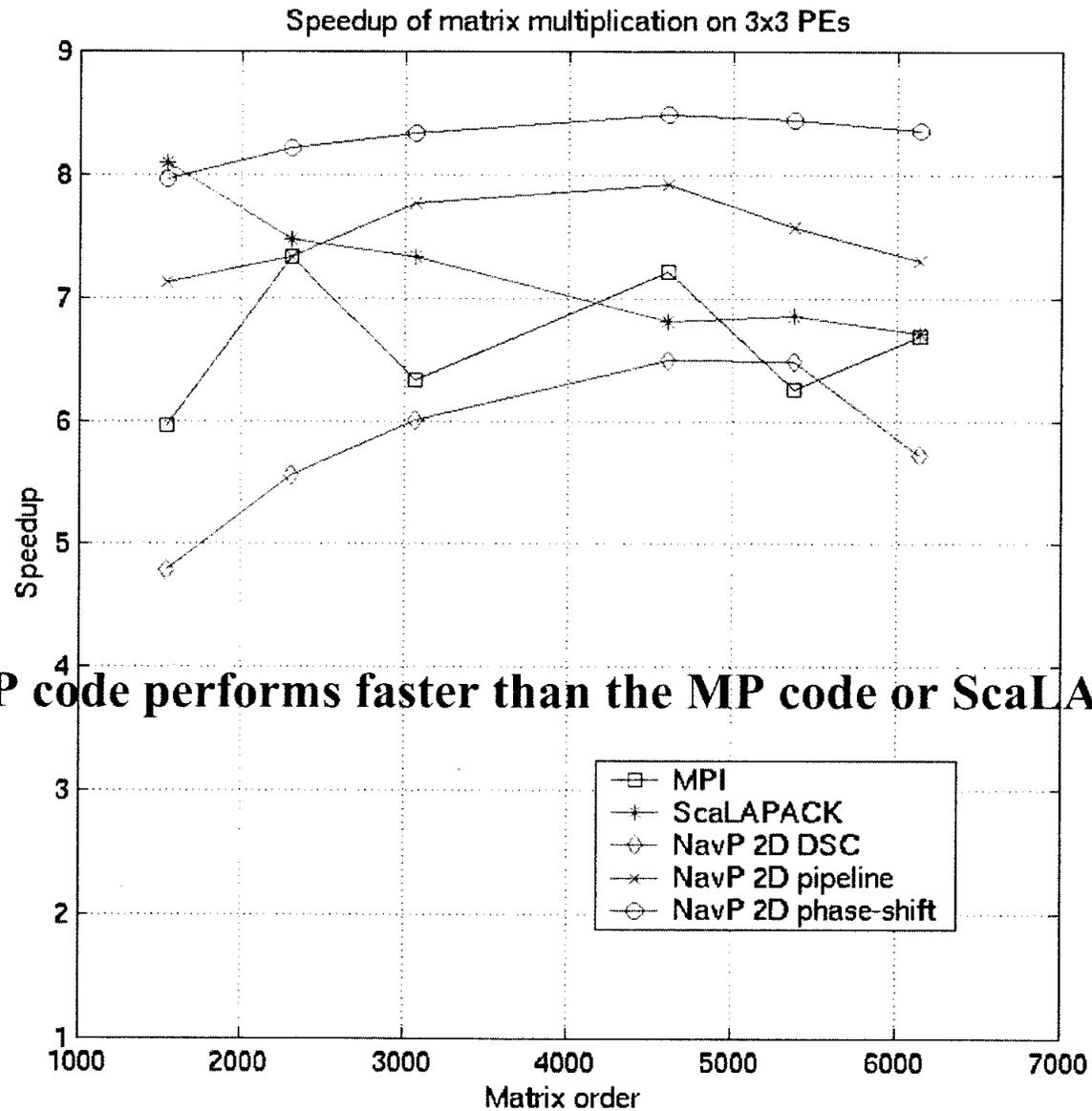
Figure 5.26. Phase shifting in both dimensions.

## Case Study 3: Matrix Multiplication (cont'd)

- NavP is amenable to incremental parallelization
  - DSC
  - Pipelining : [MatrixMult1.swf](#)
  - Phase shift: [MatrixMult2.swf](#)
  - 2-D DSC
  - 2-D Pipelining: [MatrixMult3.swf](#)
  - 2-D Phase shift: [MatrixMult4.swf](#)
- Final NavP implementation fast
- Intermediate codes have good performance and hence useful as end products
- A mechanical process that provides simple incremental steps
  - Gentleman's Algorithm was an abrupt jump

## Case Study 3: Matrix Multiplication (cont'd)

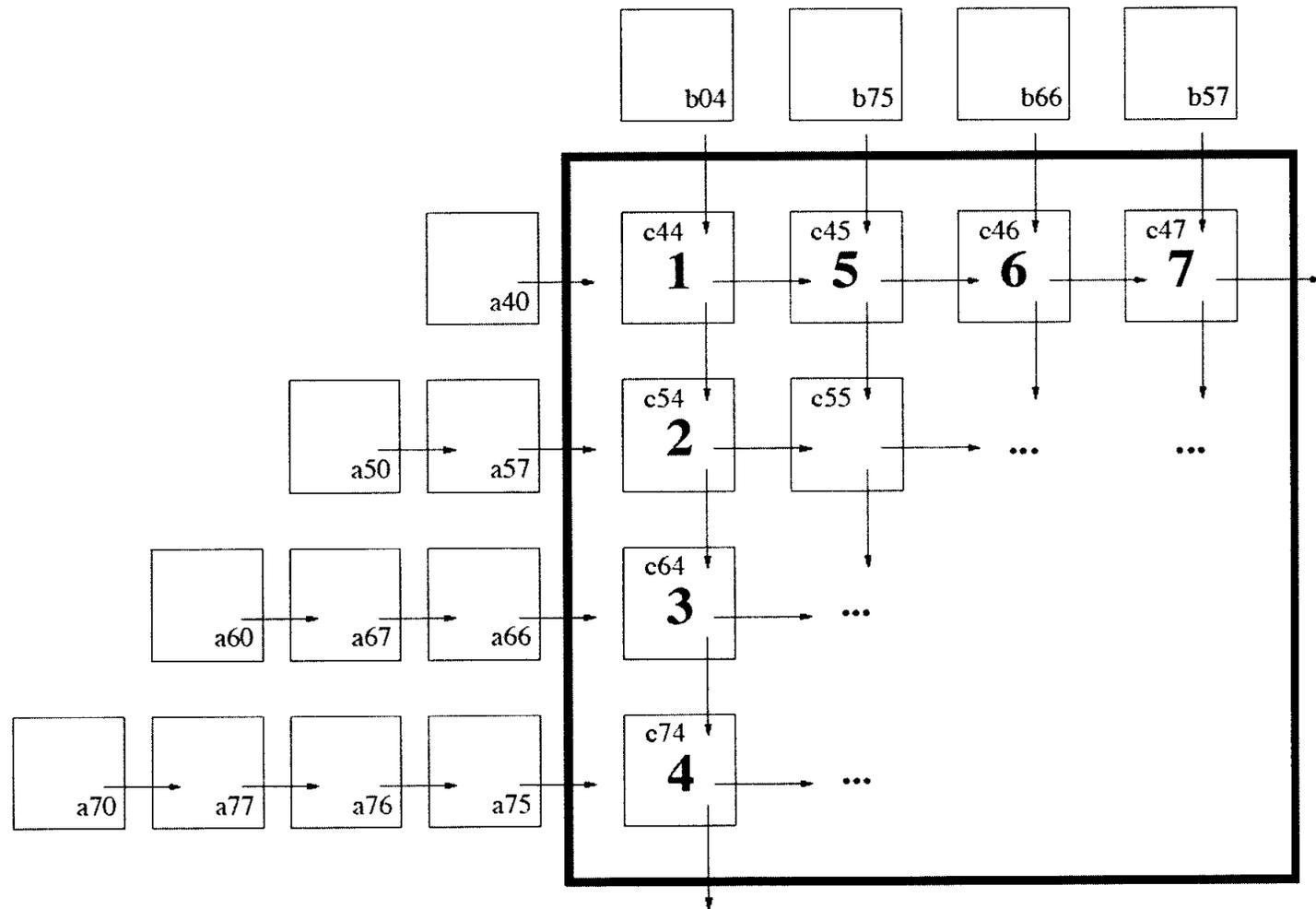
SUN workstations  
with 100 Mbps



**The NavP code performs faster than the MP code or ScaLAPACK.**

## Case Study 3: Matrix Multiplication (cont'd)

- Reason for NavP superior performance
  - multithreading (functionality factored out from application code)



## Case Study 3: Matrix Multiplication (cont'd)

- “Local services” factored out from applications and put into daemon
  - multithreading
  - self-hopping (in contrast to self-sending)
- Analogy: business travelers vs. hotel managers
  - use the NavP view for travelers
  - use the SPMD view for hotel managers
  - local services are managers’ job

## Summary: Case Studies

- Presented three case studies: Crout factorization, Cholesky factorization, matrix multiplication
- The NavP implementations are as fast or faster than the MP implementations
  - especially true when both coarse level parallelism (across nodes) and fine grained parallelism (multithreading) are needed
- Mobile pipelines are able to easily parallelize algorithms that are difficult for conventional approaches
- NavP is amenable to incremental parallelization
- NavP transformations are mechanical and possibly automatable

## Outline

1. Introduction
2. The NavP view vs. the SPMD view
3. Distributed sequential computing (DSC)
4. The methodology of NavP
5. Case Studies
6. **The enabling technology of NavP**
7. Comparisons, conclusions, and future work

## Section 6: Enabling NavP

- MESSENGERS: developed at UC Irvine
- The daemon system
- The compiler
- Data distribution for migrating computations

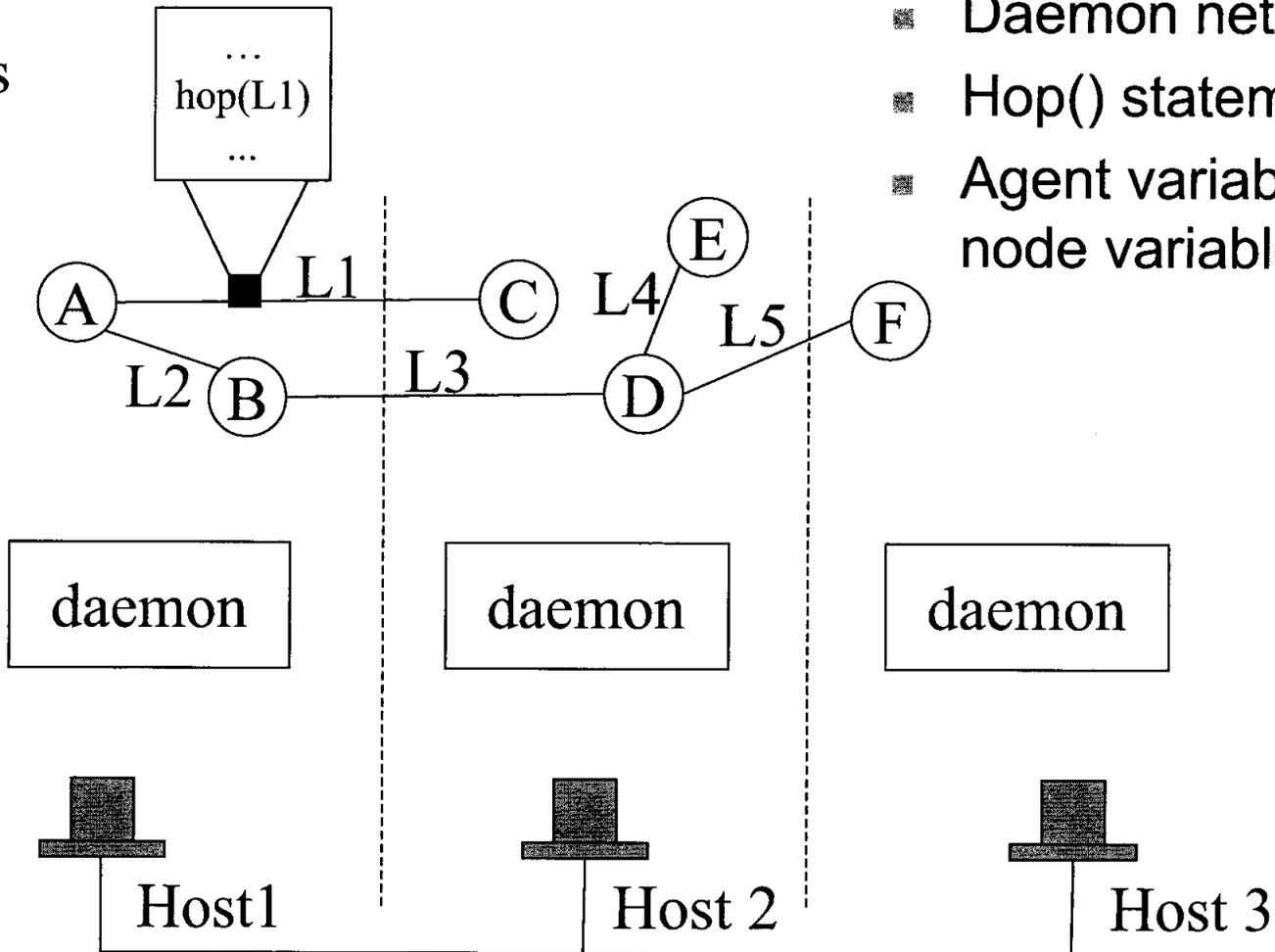
# Enabling NavP: The Daemon System

Messengers

Logical Network

Daemon Network

Physical Network

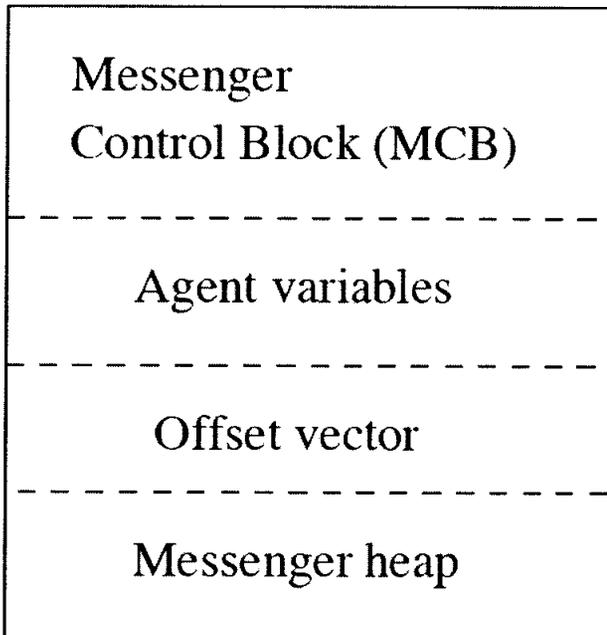


- Developed at UCI
- Daemon network
- Hop() statement
- Agent variables vs. node variables

## Enabling NavP: The Daemon System (cont'd)

- **Queues:**
  - ready queue
  - event queues
  - communication queues
  - injection queue
- **User level multithreading:**
  - Queuing operations transparent to users
  - waitEvent(evt)
    - put calling agent to event queue, if evt not signaled
    - calling agent continues, if evt already signaled
  - signalEvent(evt)
    - move waiting agents to ready queue

## Enabling NavP: The Daemon System (cont'd)



- **MESSENGERS Overhead Over MPI**
  - MCB size: 220 bytes
  - Marshalling and demarshalling
  - Context switching among agents (user level multithreading)

Figure 6.1. Messenger structure.

## Enabling NavP: The Daemon System (cont'd)

- A design based on DSM
- Use DSM right
  - large data in local memory
  - small data on DSM
  - utilize the great consistency protocols
  - easy daemon implementation

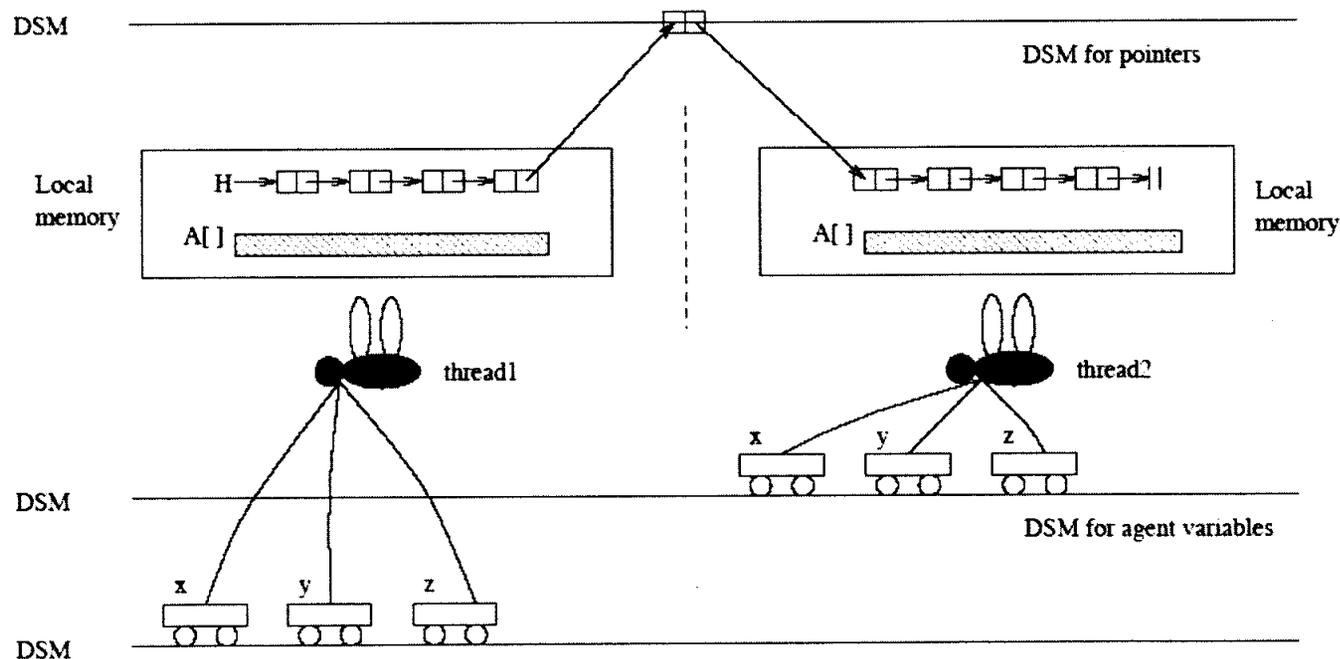


Figure 6.4. Variable assignment on the DSM-based system.

## Enabling NavP: The Compiler

- Enabling computation mobility
- Function breaking in source to source compiling
- Code does not move, computation does (with a logical program counter)
  - efficient
  - as secure as MP
- Potential problem: code size explosion

```
(1) S1
(2) hop()
(3) S2
```

(a)

```
(1) func f1(mcb)
(2)   S1
(3)   mcb->next_func = 2
(4)   ... /* code for hop() */
(5) end func
(6) func f2(mcb)
(7)   S2
(8) end func
```

(b)

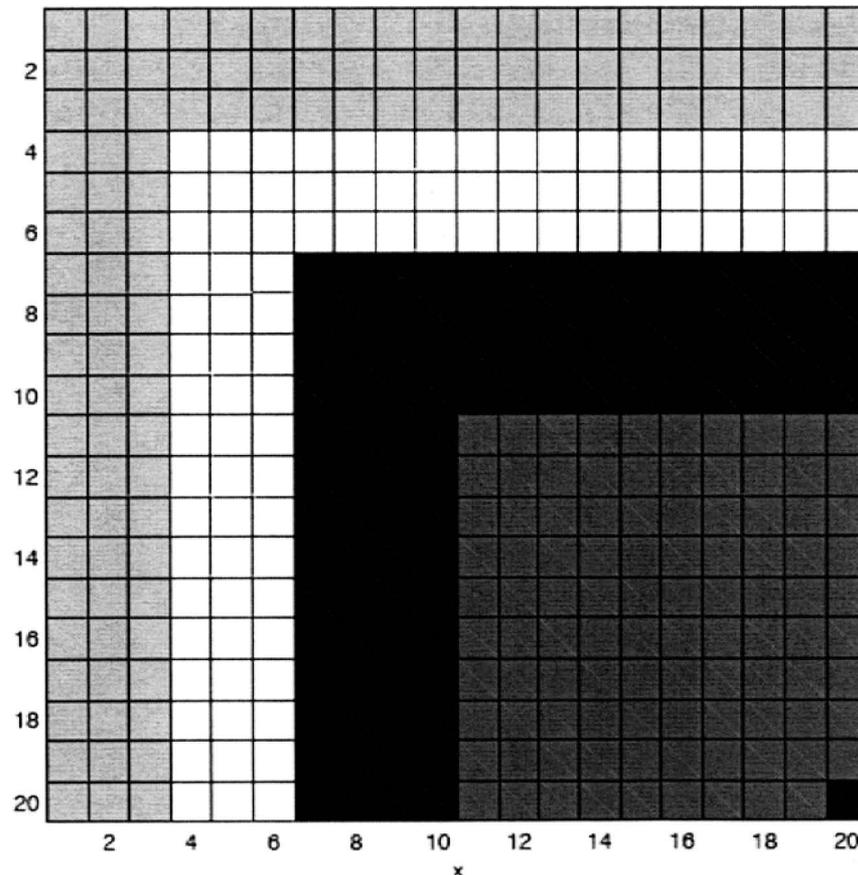
Figure 6.2. Compilation of straight-line code. (a) MESSENGERS source code. (b) Functions from source to source compilation.



## Enabling NavP: Data Distribution for Migrating Computations

- Instrument the sequential program with a small input
- Affinity multigraph:
  - Nodes are array entries
- Affinity multigraph edges:
  - an **A** edge between the LHS and a distinct RHS
  - a **C** edge between every pair of distinct RHSs
  - a **C** edge between consecutive LHSs
  - an **R** edge between neighboring array entries
- Affinity multigraph edge weights:
  - **A** edge: 1
  - **C** edge: infinitesimal (to break ties)
  - **R** edge: between 0 and 1 (to regulate the shape)
- Affinity graph partitioned (using Metis)

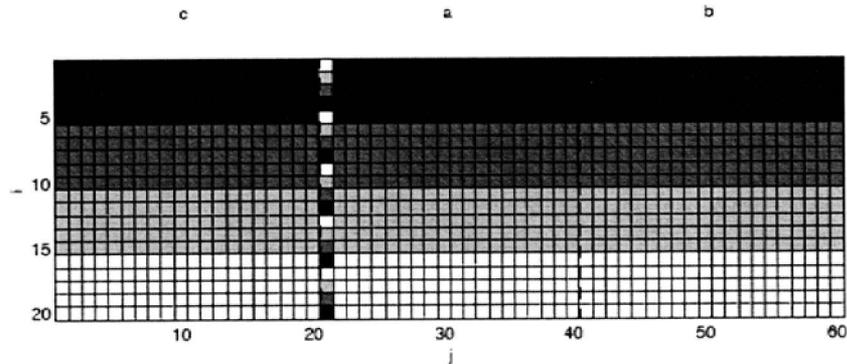
## Enabling NavP: Data Distribution for Migrating Computations



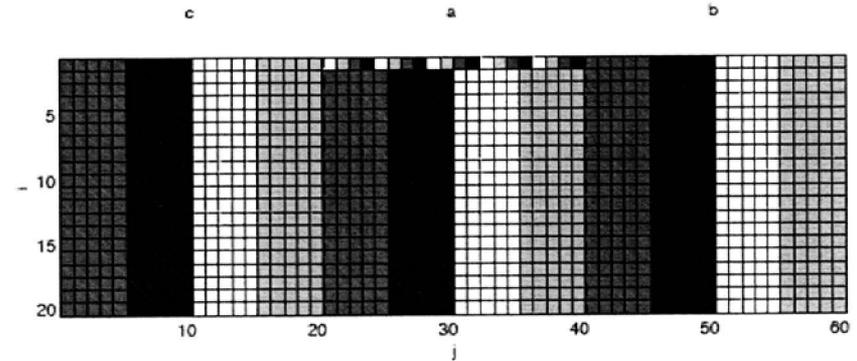
- Matrix transpose
- Each gray scale is a partition
- In contrast, conventional approaches partition the dual of a mesh
  - specific data accessing pattern of an algorithm plays no role

## Example: ADI Integration

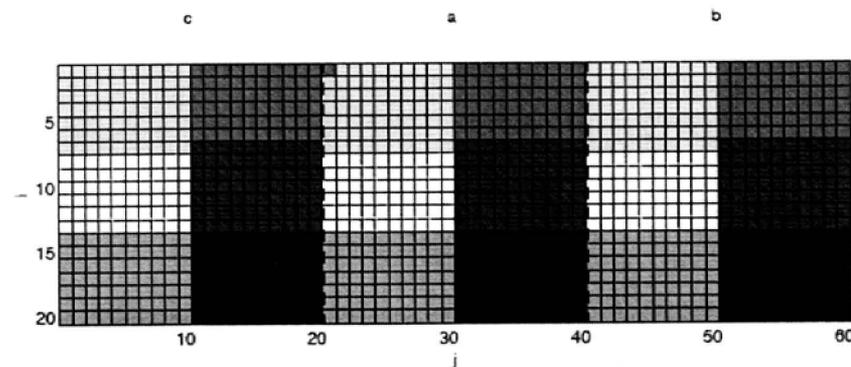
■ phase I only: row sweeping



■ phase II only: col sweeping

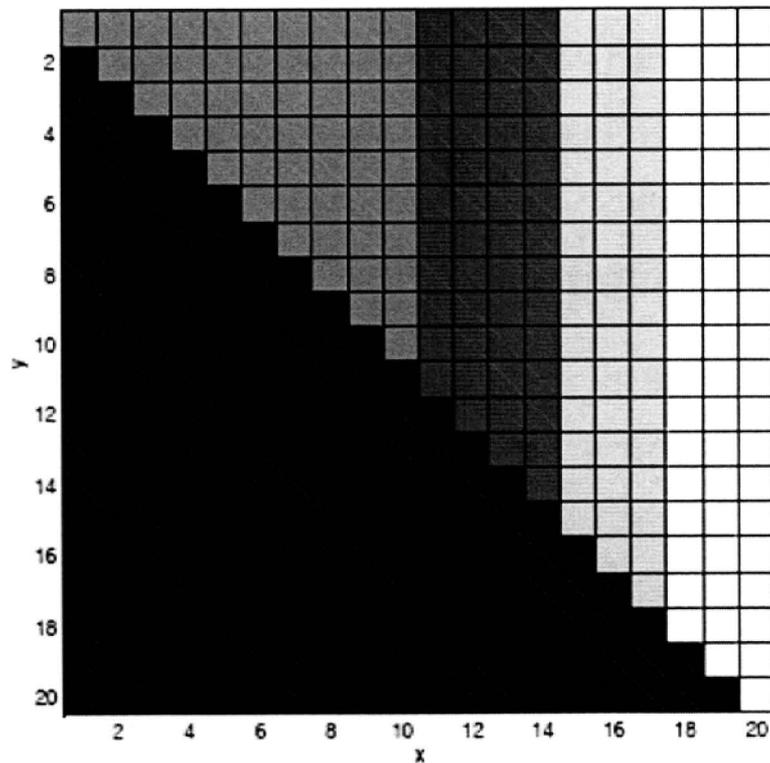


- phase I and II combined:
  - only pipeline parallelism
  - but no data redistribution in between two phases

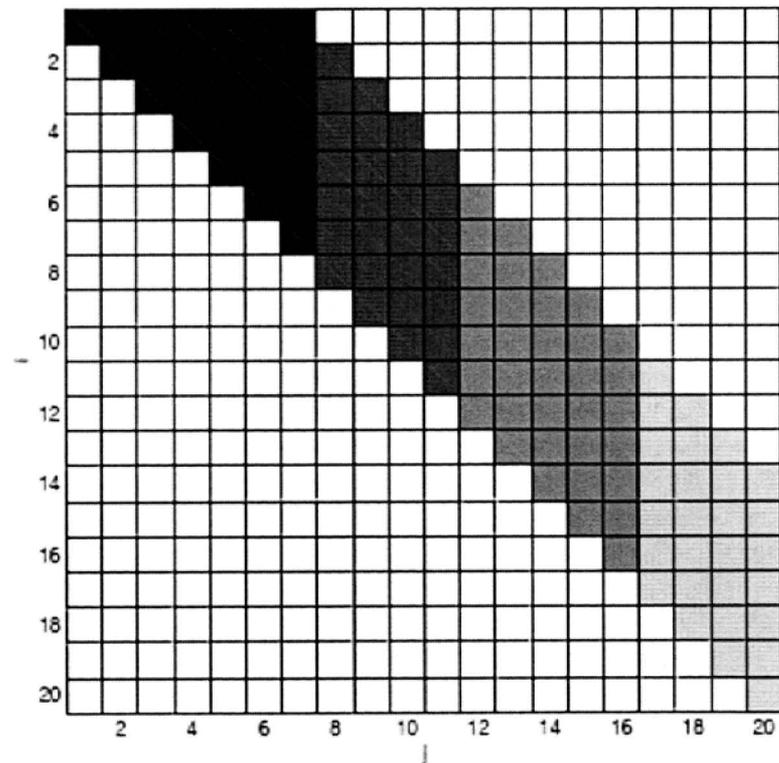


# Enabling NavP: Data Distribution for Migrating Computations

- Crout dense matrix



- Crout banded matrix



## Summary: Enabling NavP

- Presented the NavP daemon system and compiler
- Computation enabled with overhead that is insignificant in all our case studies
- Data distribution for migrating computations uses the knowledge of data accessing pattern specific to a sequential algorithm

## Outline

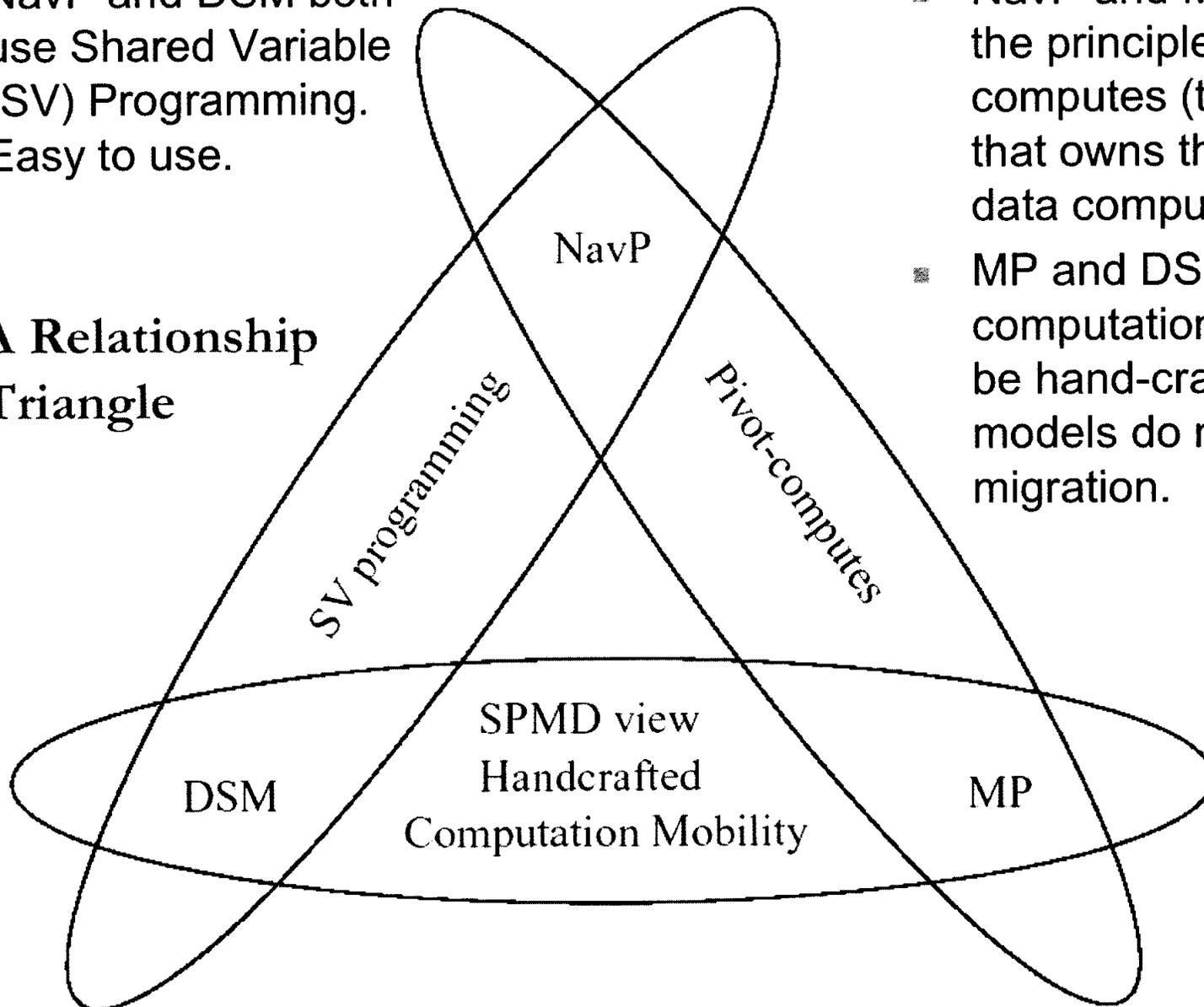
1. Introduction
2. The NavP view vs. the SPMD view
3. Distributed sequential computing (DSC)
4. The methodology of NavP
5. Case Studies
6. The enabling technology of NavP
7. **Comparisons, conclusions, and future work**

# MP or DSM versus NavP

- NavP and DSM both use Shared Variable (SV) Programming. Easy to use.

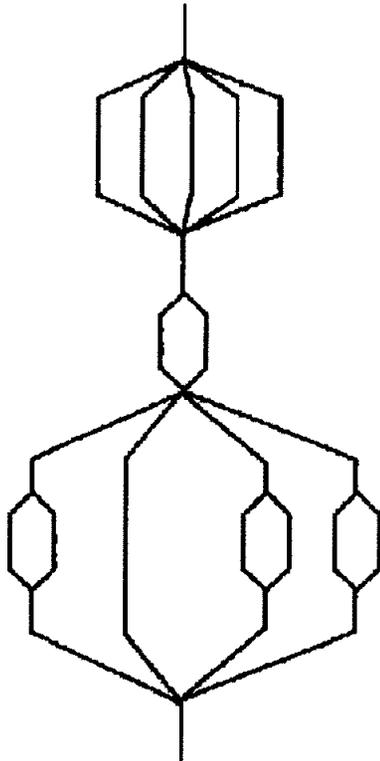
- NavP and MP both follow the principle of pivot-computes (the pivot node that owns the large sized data computes). Scalable.
- MP and DSM: computation mobility must be hand-crafted, because models do not support migration.

A Relationship Triangle



## OpenMP versus NavP

- Structured multithreading
- For shared memory (SMP)
- Uses the SPMD view
- Uses barriers

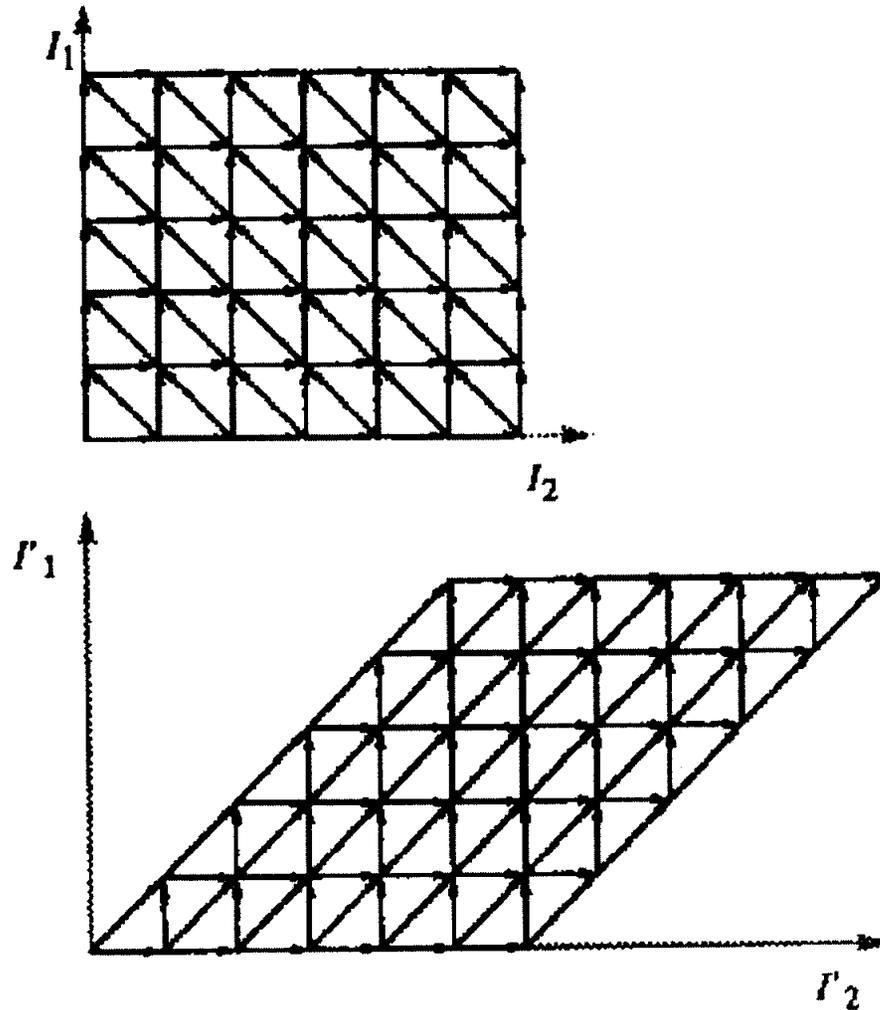


**Fig. 4.4** Structure of an OpenMP program.

```
#pragma omp parallel
{
    me = omp_get_thread_num();
    total = omp_get_num_threads();
    if (me==0)
        printf("I am the boss.");
    else if (me==1)
        specialTask();
    else
        /* Each of the tasks 2...total-1 applies
           someFunction to part of the input values,
           The input values are assigned alternately to
           the tasks such that all values are dealt with. */
        for (i = me; i < N; i += total - 2)
            someFunction(A[i]);
}
```

**Fig. 4.5** Work sharing by thread numbers.

## Parallelizing Compilers versus NavP



- Current parallelizing compilers use the SPMD view
  - skew the loop and change the code with complicated loop transformations (e.g., affine)
- NavP
  - skew the loop but still keep the original loop bounds
- NavP is not a competing technology, rather it can help parallelizing compilers to express parallelism well

## OO versus NavP

- Encapsulation puts a shield around an object
  - Object users or subtype builders do not or cannot care about the implementation details
  - Modules can be built independent of each other
  - Bug sources can be isolated
- OO view is independent of NavP view
  - NavP view degenerates on one PE, but OO view does not
  - Methods of an object will see DBlocks as distributed computation is introduced, so choosing NavP or MP is still an issue
- OO does not solve the problem rooted from distributed computing

## NavP Advantages

### ■ Performance

- Fine grained parallelism within a node, coarse grained parallelism across nodes
- NavP code performs as fast as or faster than MP code

### ■ Ease of use

- Code structure an invariant in NavP transformations
- Incremental parallelization
- New sources of parallelism (mobile pipelines)
- Backward uniprocessor compatibility
- “Local service code” factored out from application

### ■ A uniform programming model

- No need to do hybrid programming on SMP clusters
- No need to port/rewrite code for new architectures

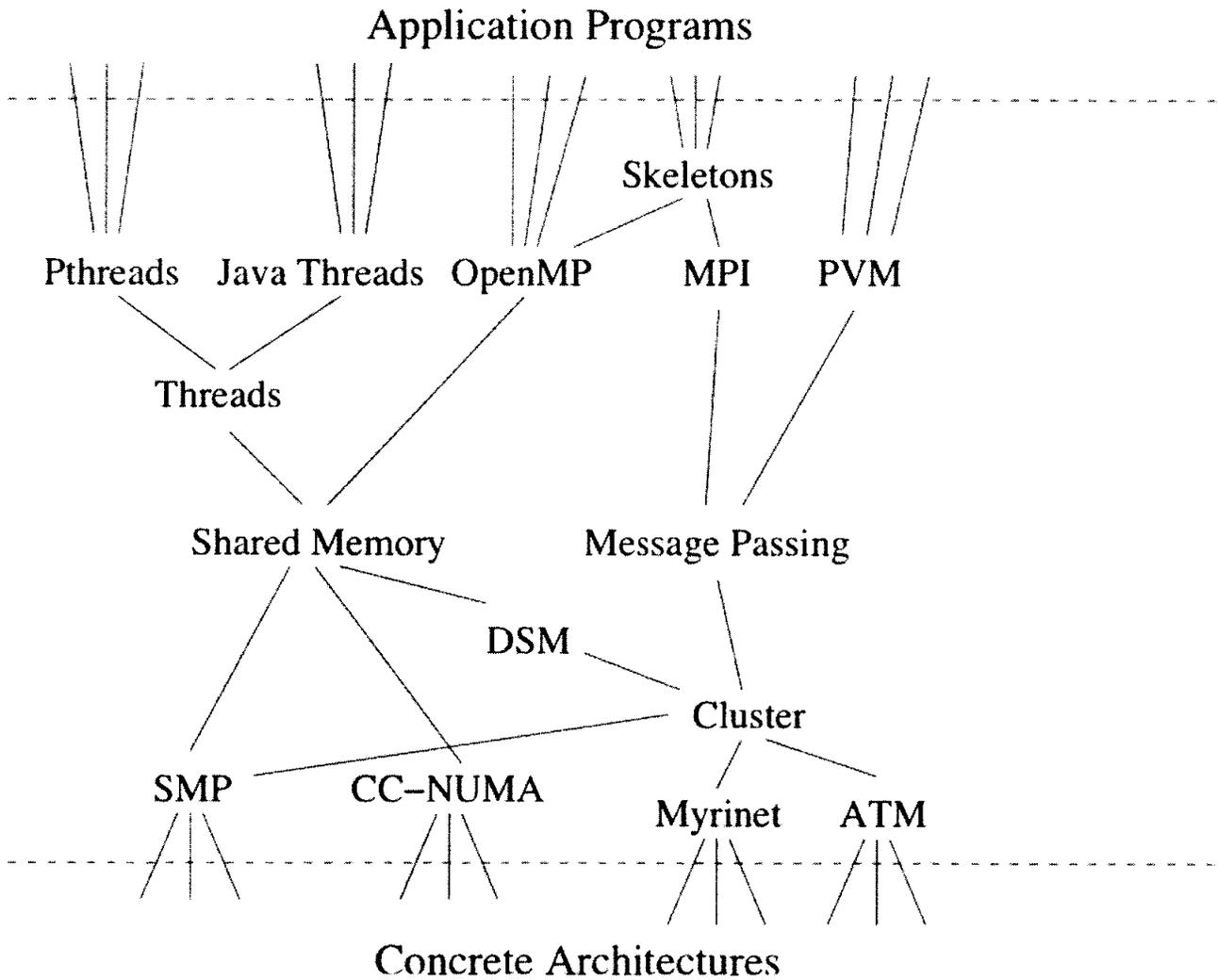
## NavP Potential Impact

- The SPMD view is popular for historical reasons
  - Computers were not widely connected
  - Programs ran on a single CPU for their lifetime
- Today single PEs is only a special case
  - The “world” is connected, and computations “flow” around
  - The MP view is no longer convenient (the “assembly language”)
- NavP to break MPI’s “monopoly”
  - MP dominates because other attempted approaches do not scale
- The usefulness of computation mobility suggests a change
  - NavP is from a very simple observation
  - But it calls for a revolution in hardware, compilers, and tools
  - Because migration is not a first-class operation at low level yet

## Summary of Results

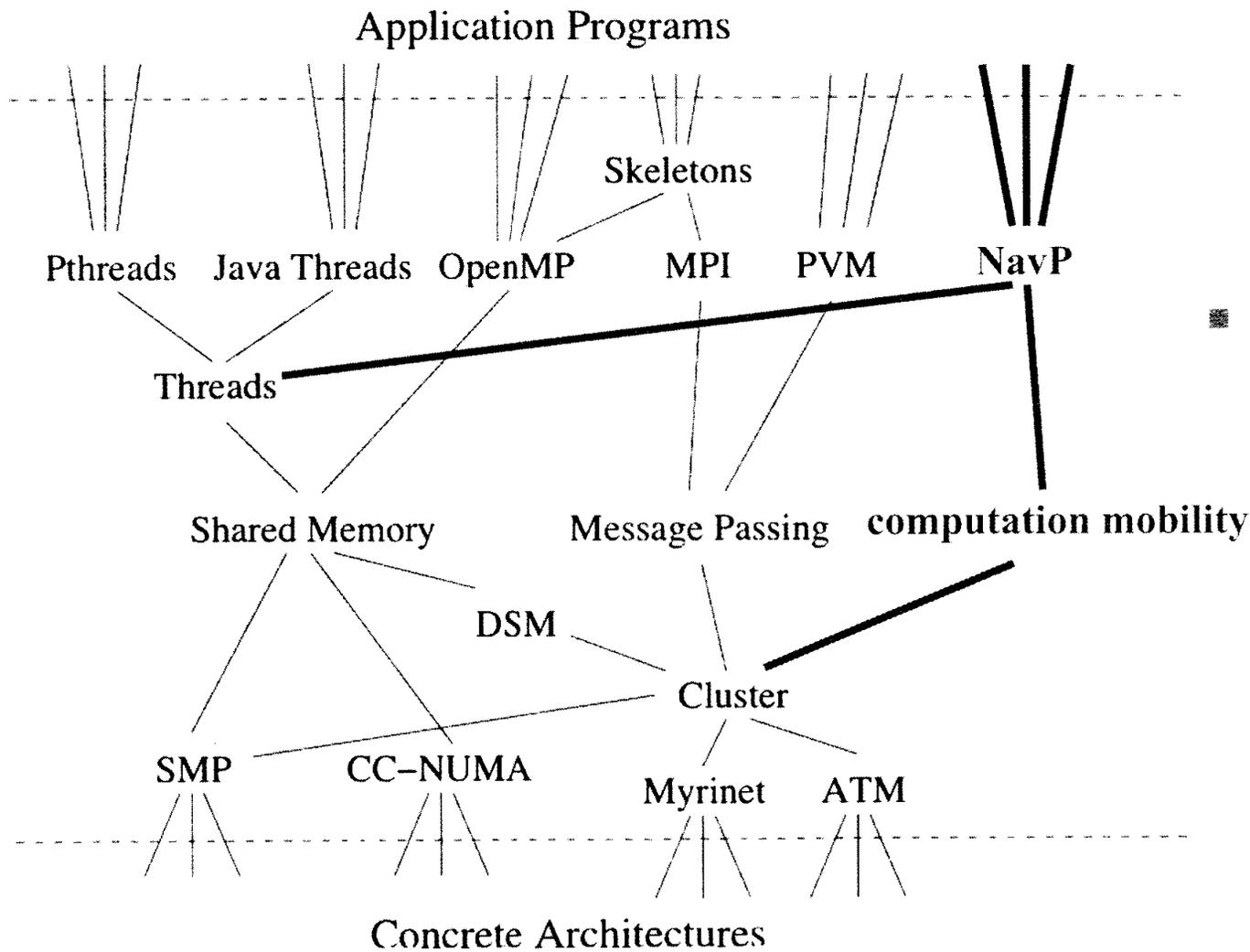
- NavP enables structured distributed programming
- NavP can parallelize algorithms that are otherwise difficult or even on surface appear impossible to parallelize
- NavP provides greatly improved programmability with negligible performance degradation
- NavP is a uniform approach
  - No more complicated hybrid programming
  - No more code rewriting for each new generation of supercomputers

# The World without NavP



(Adapted from Leopold 2001)

# NavP Contribution



(Adapted from Leopold 2001)

- Exploit both fine (multithreading) and coarse (pipelined tasks) grained parallelism
- Allow people to use the most advanced computers without being overwhelmed by the complexity of rewriting their codes with each new generation of architectures

## NavP Future Work

- Compiler for DSC (2D optimizing compiler)
  - heuristics for choosing the right DBlocks to resolve
  - spatial optimization (as opposed to temporal)
- Compiler for mobile pipelines
  - coarsening the dependency relationship to thread/task level
- NavP to/from SPMD translators
  - card dropping and card gathering
- Support computation mobility on large-scale SMP clusters
  - more efficient daemons for SMP/Cell/Multicore
  - priority queues for local and global traffic control
- New language bindings (e.g., Fortran)
  - MP is advantageous in this regard
- NavP for the Grid (a security mechanism)
- Supporting domain decomposition and ghost boundary at system level
- Automatic coarsening of communication at system level

## Caution

- NavP is still a manual programming approach as of today
- Distribution parallel programming remains an art, just as sequential programming
- NavP may help express irregular communication patterns, locality of access, and parallelism well, but the optimization of the quantities continues to be tough math problems
- Only the tip of a huge iceberg. No silver bullet!

## Comments, Questions, and Discussions

**Thank you very much!**