

CMPUT 681:
Parallel and Distributed Systems

Paul Lu

`paullu@cs.ualberta.ca`

`http://www.cs.ualberta.ca/~paullu/C681/`

Top 3 Lessons of CMPUT 681

1. Granularity.
2. Optimize for the common case.
3. Do **not** copy data or block, if you avoid it.

Motivation

- Uniprocessors are fast, but somebody always wants more!
 - Some problems require too much **computation**
 - Some problems use too much **data**
 - Aside: Some problems have too many **parameters** to explore
- For example: weather simulations, other scientific simulations, game-tree search, Web servers, databases, code breaking
 - ⇒ need good sequential algorithms
 - ⇒ need good parallel algorithms
 - ⇒ need good systems software (e.g., programming systems)
 - ⇒ need good parallel hardware

Parallel vs. Distributed Computing

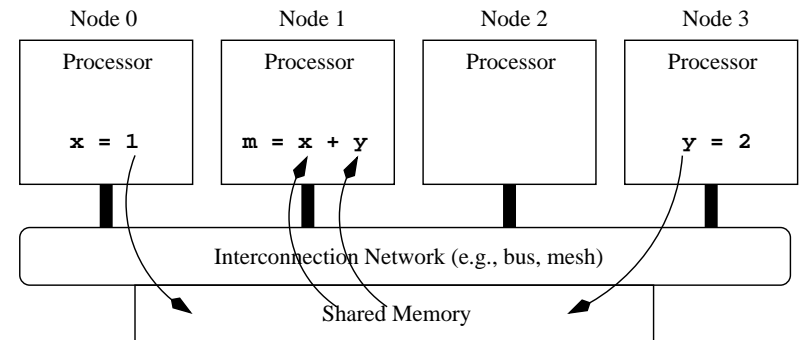
Some hardware-centric terminology:

- parallel computing \Rightarrow shared memory
 - multiprocessors (important trend)
 - processes share logical address spaces
 - processes share physical memory
 - sometimes refers to the study of parallel algorithms
- distributed computing \Rightarrow distributed memory
 - clusters and networks of workstations (NOW)
 - processes do **not** share address spaces
 - processes do **not** share physical memory
 - sometimes refers to the study of theoretical distributed algorithms or neural networks

Parallel Algorithms: The Basics

- A parallel algorithm solves a specific problem by dividing the computation into smaller units of work that can be solved concurrently and then combined to form the final answer.
- The basic idea:
 1. Create work
 - partition, divide, embarrassingly parallel, granularity
 2. Coordinate computation
 - IPC, synchronization, load balancing, synchronous/asynchronous communication, dependency
 3. Combine results
 - termination detection
 4. Repeat for all “work”
- **The number one lesson of C681:** Granularity – the ratio between computation and communication / synchronization / coordination

Shared Memory

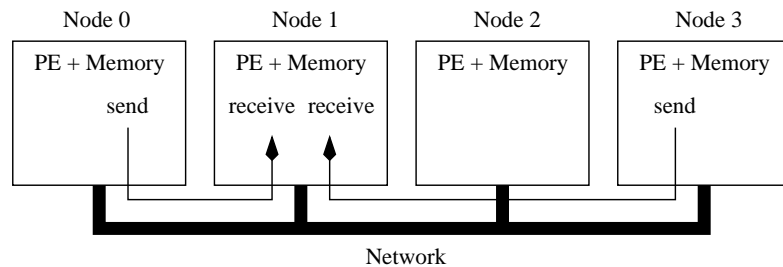


- hardware-based shared memory with OS support

Two example systems:

1. SGI Origin 200 (charron.cs)
 - 8×350 MHz R12000, 8 GB
2. SGI Origin Model 3900 (“arcturus”)
 - 256×700 MHz R16000, 256 GB RAM

Distributed Memory



- cannot directly and asynchronously read remote memories
- explicit **sends** and **receives** to share data (e.g., MPI, PVM)
- fast network (e.g., Myrinet, Gigabit Ethernet, ATM, InfiniBand, Quadrics)
- For example, Calgary's clusters and AICT, etc.
 - Quadrics-based in Calgary

A Case for NOW (1995)

- Cost advantage through large volumes
 - Ideas/design move from high to low end (e.g., FPU, graphics, vector processing, gigabit networking)
 - Once the mass market adopts the technology, watch out! Any exceptions?
 - Future: Impact of consumer electronics (e.g., Playstation 3, iPhone/iPod)
- Switching to commodity parts helps, but..
 - Engineering delays to integrate
 - Time delays \implies inferior performance
 - Today: Clusters, Blades, Myrinet SANs
- Software is **still** the weak link
 - Avoiding N.I.H. syndrome is good
 - Today: Free OSES (Linux, *BSD, open source) changing the landscape
 - Future: Shrink-wrapped parallel applications (e.g., DB2, parallel Matlab)

Flynn's Taxonomy (1966)

	Single Data Stream	Multiple Data Stream
Single Instruction Stream	SISD (e.g., uniprocessor)	SIMD (e.g., vector processors, multimedia extensions)
Multiple Instruction Stream	MISD (mostly nonsense)	MIMD (e.g., SMP, NOW)

- MIMD is the most general-purpose
 - multiple program multiple data (MPMD): client-server, etc.
 - single program multiple data (SPMD): our focus!

Speedup

How do we characterize or measure performance?

- Let t_1 be the time to solve the problem sequentially. NOTE: Different from textbook!
- Let t_p be the time to solve the problem in parallel using p processors
- Then, speedup $S(p)$ for problem size n is defined as:

$$S(p) = t_1/t_p$$

- $S(p) = p \implies$ linear speedup or unit-linear speedup or ideal speedup. This is rare!
- $S(p) < p \implies$ sublinear speedup. Common!
- $S(p+1) = S(p) + \delta$, where $\delta < 1 \implies$ diminishing returns, especially if $\delta \ll 1$
- $S(p+1) < S(p) \implies$ slowdown
- $S(p) > p \implies$ superlinear speedup. This is largely fallacy.

Speedup scenarios

- $S(p) = p$ (unit-linear)
 - Embarrassingly parallel with low communications overheads
- $S(p) < p$ (sublinear) and $S(p+1) = S(p) + \delta$, where $\delta < 1$ (diminishing returns)
 - Idle time due to load imbalance
 - Overhead due to communication, etc.
 - Idle time due to synchronization
 - Extra computation in parallel program
 - And many more reasons.
- $S(p+1) < S(p)$ (slowdown)
 - Overheads are $O(p)$ “but” work is $O(n)$
 - Contention for a resource depends on p

Speedup Skepticism

Is superlinear speedup ($S(p) > p$) possible? Yes and no.

- Often, increasing p also increases the amount of cache memory (or even main memory), which is unfair to the sequential case.
- For certain search algorithms, a cutoff or “eureka jump” can cause superlinear speedup.
- If neither of these two explanations apply, ask some hard questions about the sequential algorithm.
- But, pragmatically, we are happy to accept superlinear speedup if it happens.

Bottom line: Do not rely on a single number to tell the full story.

Granularity

Granularity is an informal concept with (at least) two main definitions:

1. The amount of computation that typically occurs between communication or synchronization points.
2. The ratio of computation to communication.

Granularity can be changed by:

- Improving the algorithm to require less or cheaper communication
- Reducing the cost of communication
- Reducing the amount of synchronization: reduce contention and reduce idle time
- Increasing the size of the problem
- What if the communication overheads are $O(n^2)$ but computation is $O(n^3)$?

Amdahl's Law

In 1967, Gene Amdahl argued that the inherently sequential portions of a parallel program will dominate its speedup performance.

- Let *seq* be the portion of a program's execution time that is inherently sequential. Examples?
- Let *para* be the portion that is parallelizable, where *total time* = *seq* + *para* = 1, for simplicity
- In the ideal case, we can achieve unit-linear speedup for the parallel portion. Therefore:

$$S(p) = \frac{seq+para}{seq+para/p} = \frac{1}{seq+para/p}$$

- And

$$\lim_{p \rightarrow \infty} S(p) = \frac{1}{seq}$$

SPMD Programming

Recall that SPMD is one approach to MIMD.

There are usually 3 main parts to a SPMD program.

1. Initialize data structures (and start threads)
 - Under Pthreads, start-up is single threaded.
Must create threads.
 - Under MPI, start-up is multi-process.
2. SPMD execution
3. Clean up and exit

Creating Threads

```
struct ThreadControlBlock
TCB[ NUM_THREADS ];
pthread_t ThreadID[ NUM_THREADS ];
void * mySPMDMain( void * );

int main( int argc, char ** argv )
{
    /* Initialize global data here */
    /* Start threads */
    for( i = 1; i < NUM_THREADS; i++ )
    {
        TCB[ i ].id = i;      /* In parameter */
        pthread_create( &(amp; ThreadID[ i ] ), NULL,
                        mySPMDMain, (void*)&( TCB[ i ] ) );
    }
    TCB[ 0 ].id = 0;
    mySPMDMain( (void*)&( TCB[ 0 ] ) );

    /* Clean up and exit */
}
```


SPMD Execution

```
#define MASTER    if( myId == 0 )
#define BARRIER barrier( myId, __LINE__ );

void * mySPMDMain( void * arg )
{
    struct ThreadControlBlock * myTCB;
    int myId;
    pthread_t * myThreadIdPtr;

    /* Actual parameter */
    myTCB = (struct ThreadControlBlock *)arg;

    /* Other parameters passed in via global */
    myId = myTCB->id;

    /* Parallel array to TCB */
    myThreadIdPtr = &(amp; ThreadID[ myId ] );

    ... continued ...
```

SPMD Execution (cont.)

```
BARRIER;
startTiming();

/* Phase 1 */
BARRIER;

/* Phase 2 */
MASTER
{
}
BARRIER;

/* Phase 3 */
BARRIER;

/* Phase 4 */
BARRIER;

stopTiming();
} /* mySPMDMain */
```

Non-Determinism

In the absence of very fine-grained synchronization, it is possible that instructions will be interleaved in a pseudo-random order.

Timer interrupts (i.e., end of time quantum), I/O, cache coherence actions, contention for resources, other users, etc. can all cause non-determinism.

The implications are:

1. Bugs may be hard to re-produce.
2. Slight variations in (real time) timings.
3. Depending on the algorithm and dataset, you can get different answers on different runs (e.g., floating point computations)

A related concept to non-determinism is a “race condition.”

Solution? Synchronize properly/more. But, be careful of how that impact performance.

Parallel Sorting by Shan and Singh

My comments so far:

- Sorting is a fundamental problem and widely studied
- Sorting can be highly architecture-specific. Many sorting algorithms are too fine-grained to be practical on real machines.
- Sorting algorithms differ in their:
 1. Granularity
 2. Number of messages required
 3. Size of messages required
- Therefore, an algorithm that is good for one architecture is not necessarily good on another architecture
- Tradeoffs can be made between extra computation and cheaper communication
- The specific programming model affects how an algorithm is implemented

DSM vs. DSD

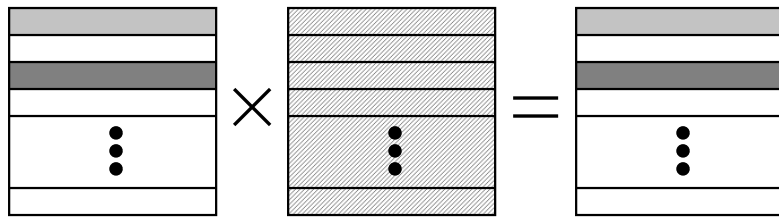
	DSM	DSD
What is shared?	Memory in an address space	Data structures
Naming	Pointer	ADT and pointer
Mechanisms	Page faults	ADT and interfaces
Unit of management	Fixed: page <div><div>x</div><div>y</div> or <div><div>x</div><div>y</div></div></div> <div><div>x</div><div>y</div></div>	Variable: object or region <div><div>x</div><div>y</div></div>
Can suffer from false sharing?	Yes	No

DSM vs. DSD (2)

	DSM	DSD
Unit of management	Fixed: page <div><div>x</div><div>y</div> or <div><div>x</div><div>y</div></div></div> <div><div>x</div><div>y</div></div>	Variable: object or region <div><div>x</div><div>y</div></div>
Unit of sharing policy	All shared pages; sometimes per-page	Object or region
Can alter sharing policy on per-context basis?	Possible, but...	Yes

Data Sharing Patterns

```
// Sequential matrix multiplication
for( i = 0; i < size; i++ )
  for( j = 0; j < size; j++ )
    mC[i][j] = dotProd( &mA[i][0], mB, j, size );
```

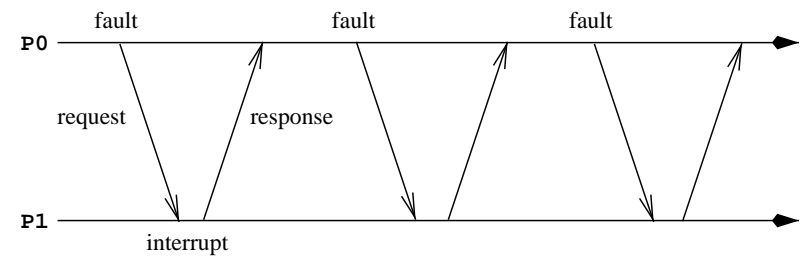


Matrix A Matrix B Matrix C

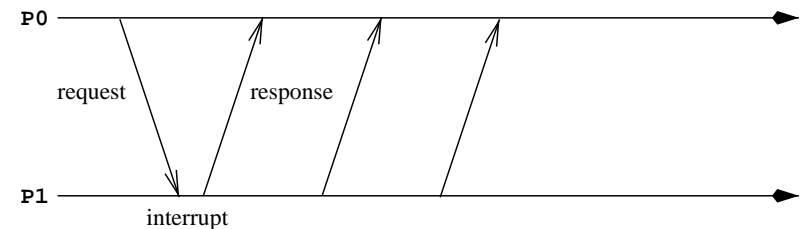
1. Matrix A is read-only; rows are independent
 2. Matrix B is read-only; large working set
 3. Matrix C is write-only
- What if, later on, we want $B \times C = A$?

Optimizing Patterns: Bulk Data

- Consider Matrix B
- Basic DSM faults and transfers each page individually

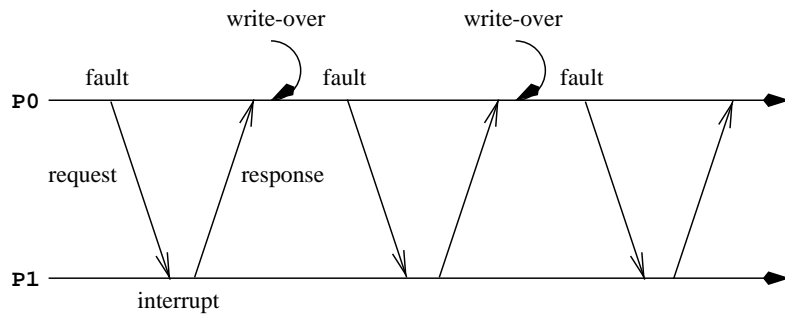


- If possible, better to do a bulk-data transfer

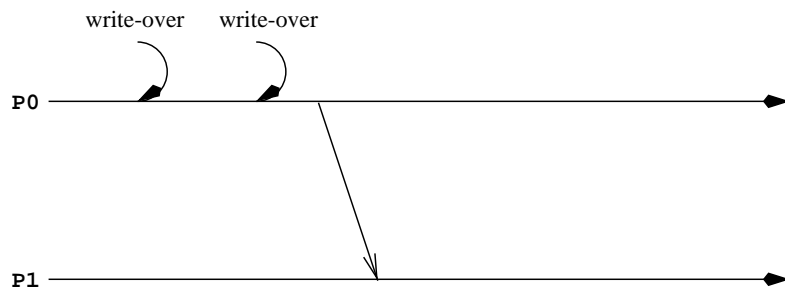


Optimizing Patterns: Write-Only

- Consider Matrix C
- Basic DSM faults and transfers each page before writing over



- If possible, avoid paging-in for write-only



Optimization Flexibility

Observations:

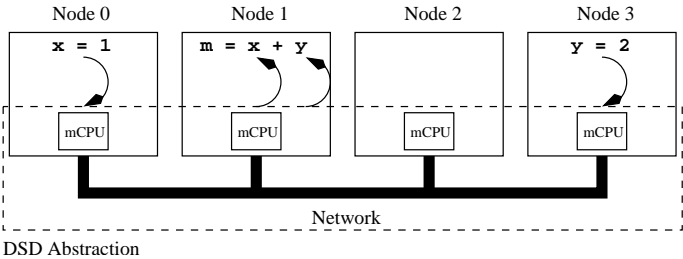
- Can optimize for data sharing patterns
- Data sharing patterns can change from phase to phase (i.e., $A \times B = C$ then $B \times C = A$)

⇒ reduce number of messages and bytes sent

Issues:

- Which mechanisms are useful?
- How to implement? Pre-processors? Compiler #pragmas? Run-time flags?
- What algorithms and strategies are useful?

Scoped Behaviour and Aurora



- 1. **Ease of use:** High-level abstract data types (ADT) for shared data using C++ objects
- 2. **Implementation:** Class hierarchy and novel *scoped behaviour* approach
- 3. **Performance:** Optimizations that exploit semantics about data-sharing patterns

Layered View of Aurora

Layer	Main Components and Functionality
Programmer's Interface	Process models Distributed vector and scalar objects <i>Scoped behaviour</i>
Shared-Data Class Library	Handle-body shared-data objects Scoped handles implementing data-sharing optimizations
Run-Time System	Active objects and remote method invocation (currently, ABC++) Threads (currently, POSIX threads) Communication mechanisms (shared memory, MPI, UDP sockets)

Example: Simple Update Loop

Shared data as objects:

```
GVector<int> vector1( vsize );
```

Original code:

```
for( i = 0; i < vsize; i++ )
    vector1[ i ] = someFunc();    // Context 1

vector1[ 0 ] = 1;                // Context 2
```

*With scoped behaviour
(i.e., programmer annotations):*

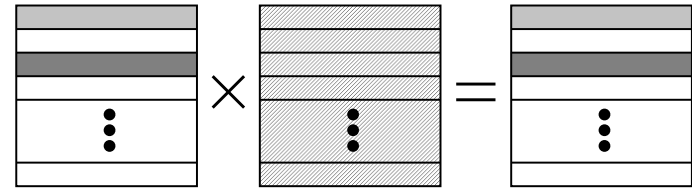
```
{ // Begin scope
    NewBehaviour( vector1, GVReleaseC, int );

    for( i = 0; i < vsize; i++ )
        vector1[ i ] = someFunc();    // Context 1
} // End scope

vector1[ 0 ] = 1;                // Context 2
```

Example: Matrix Multiplication

Scoped behaviour is a flexible programmer's interface to system-provided data-sharing optimizations.



```
GVector<int> mA; GVector<int> mB; GVector<int> mC;

{ // Begin scope
    NewBehaviour( mA, GVOwnerComputes, int );
    NewBehaviour( mB, GVReadCache, int );
    NewBehaviour( mC, GVReleaseC, int );

    while( mA.doParallel( myTeam ) )
        for(i = mA.begin(); i < mA.end(); i += mA.step())
            for( j = 0; j < size; j++ )
                mC[i][j] = dotProd( &mA[i][0], mB, j, size);
} // End scope
```

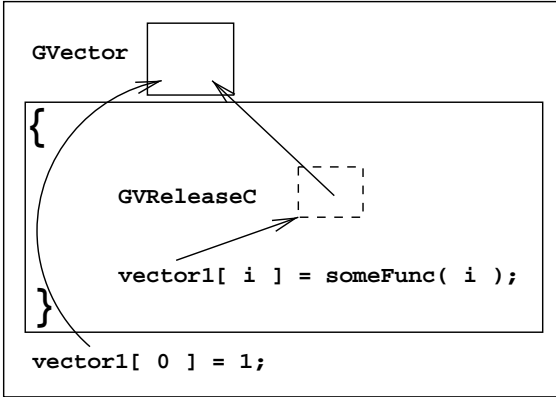
Implementing Scoped Behaviour

```
GVector<int> vector1( vsize );

{ // Begin scope
  GPortal<GVector<int> > AU_vector1( vector1 );
  GVReleaseC<int> vector1( AU_vector1 );

  for( i = 0; i < vsize; i++ )
    vector1[ i ] = someFunc( i );
} // End scope

vector1[ 0 ] = 1;
```



An Implementation Framework

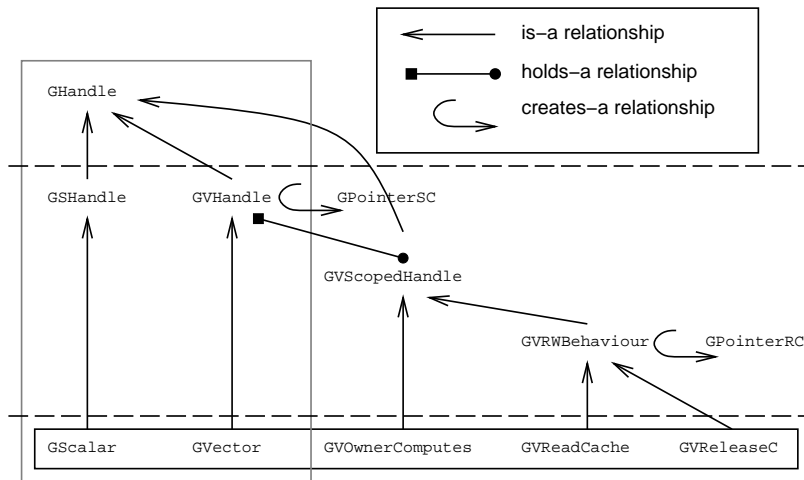
Scoped behaviour is a change in the interface or implementation of an ADT for the lifetime of a language scope.

Scoping View: Begin scope—In scope—End scope

	GVector	Scoped Behaviour
		GVReleaseC
Begin scope (Constructor)	Create shared-data objects	Create update buffers
In scope (operator[])	Immediate data access	Buffer updates, synchronous reads
End scope (Destructor)	Delete objects	Flush and free buffers

- Within the framework, a number of optimizations can be implemented
- High-level semantics of the behaviour can be exploited at various software layers

Class Hierarchy for Handles



- Focus on inheritance and operator overloading
- Programmer only concerned with GScalar, GVector, GVOwnerComputes, GVReadCache, and GVReleaseC

Experimental Evaluation

1. Cluster of 16 workstations with ATM (POW)
2. Compare 3 different systems:
 - (a) **Aurora** (DSD, scoped behaviour)
 - (b) **TreadMarks** (DSM)
 - (c) **MPICH** (message passing)
3. Compare using 4 different applications (and 10 different datasets)
 - (a) Matrix multiplication
 - (b) 2-D diffusion
 - (c) Parallel sorting (PSRS)
 - (d) Travelling salesperson (TSP)

Characteristics of Applications

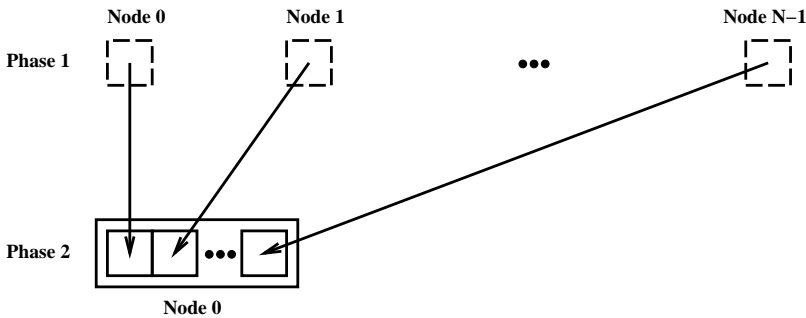
Characteristic	MM2	2DD	PSRS	TSP
Communication Intensive	Yes	No	Yes	No
Main data-sharing pattern(s)	Allgather	Neighbour	Broadcast, Gather, Alltoall (vector variant)	Master-worker, eager update of scalar
Explicit data placement	No	No	Yes	Yes
Number of computational phases in algorithm	2	1	4	1
Output of one phase used as input of another	Yes	n/a	Yes	n/a
Static load balancing	Yes	Yes	No	No
Data parallel (SPMD)	Yes	Yes	Yes	No
Task parallel (master-worker)	No	No	No	Yes

PSRS: Producer-Consumer

```
GVector<int> Sample( vsize, Nodes( 0 ) );
```

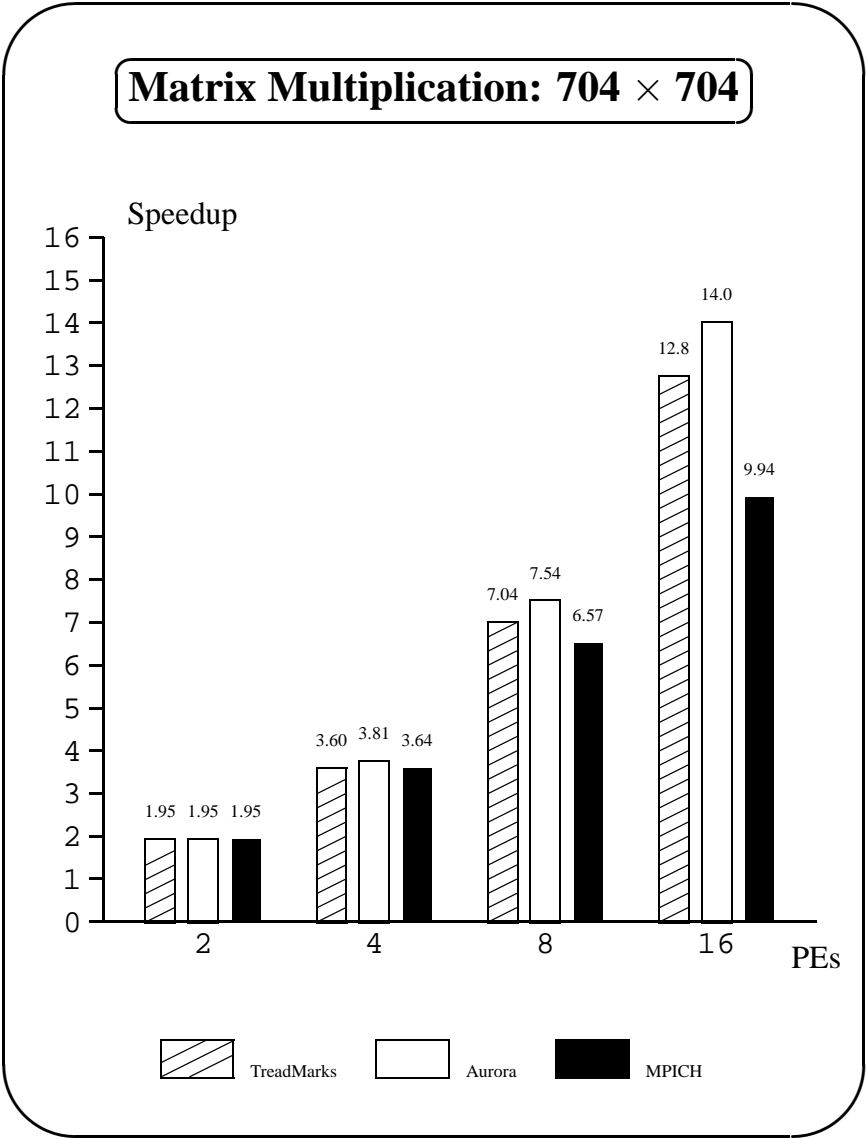
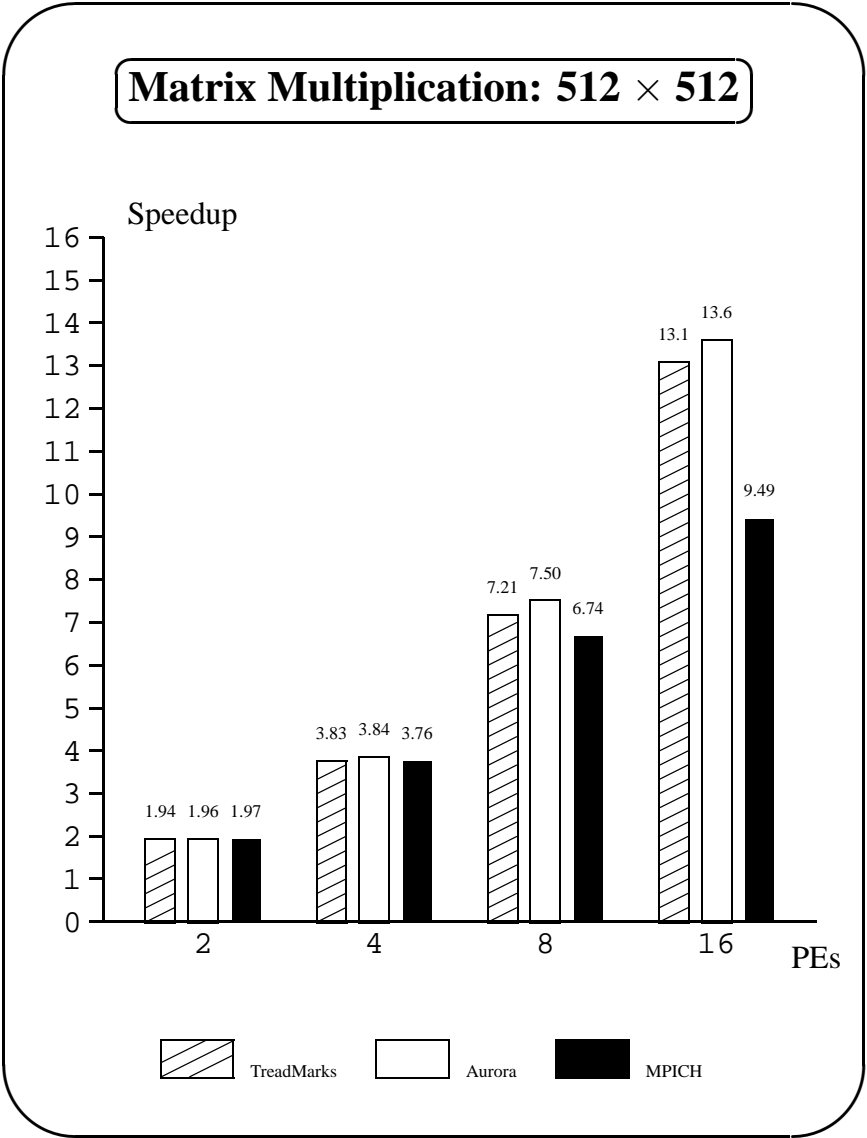
Phase 1: All nodes are *producers* and use:

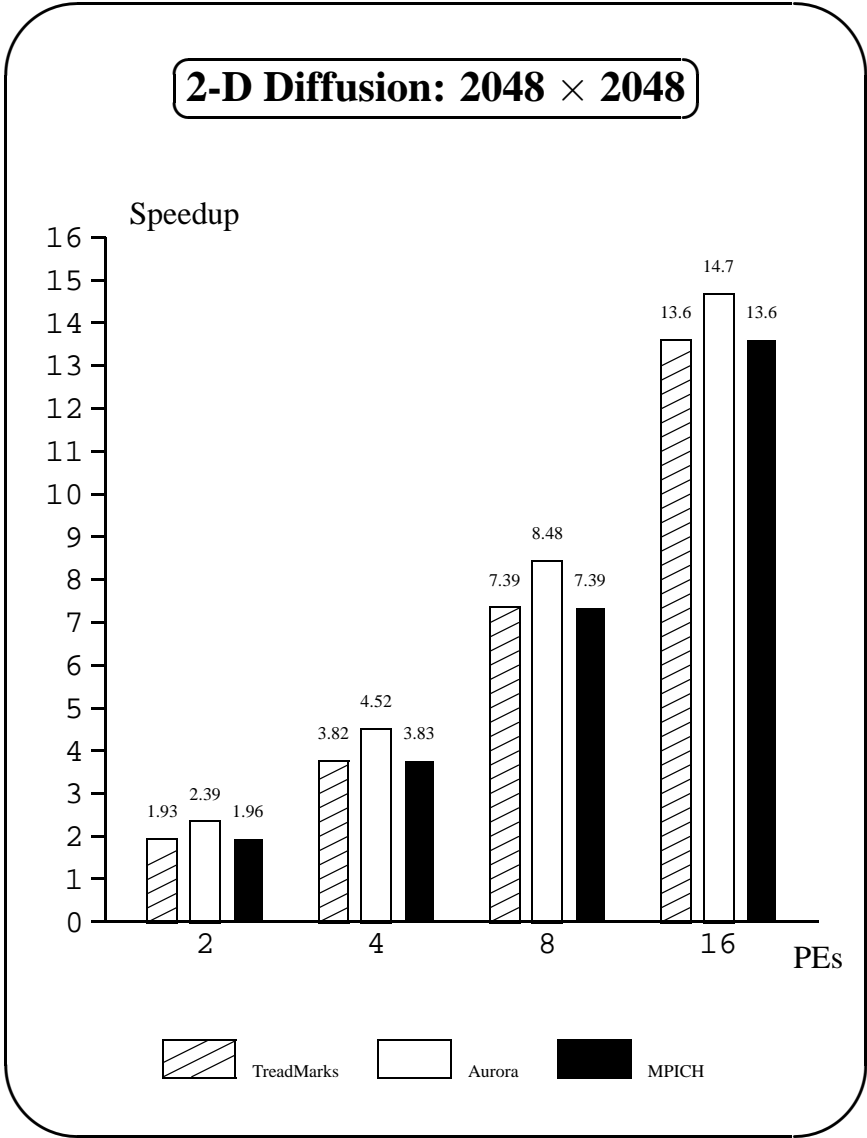
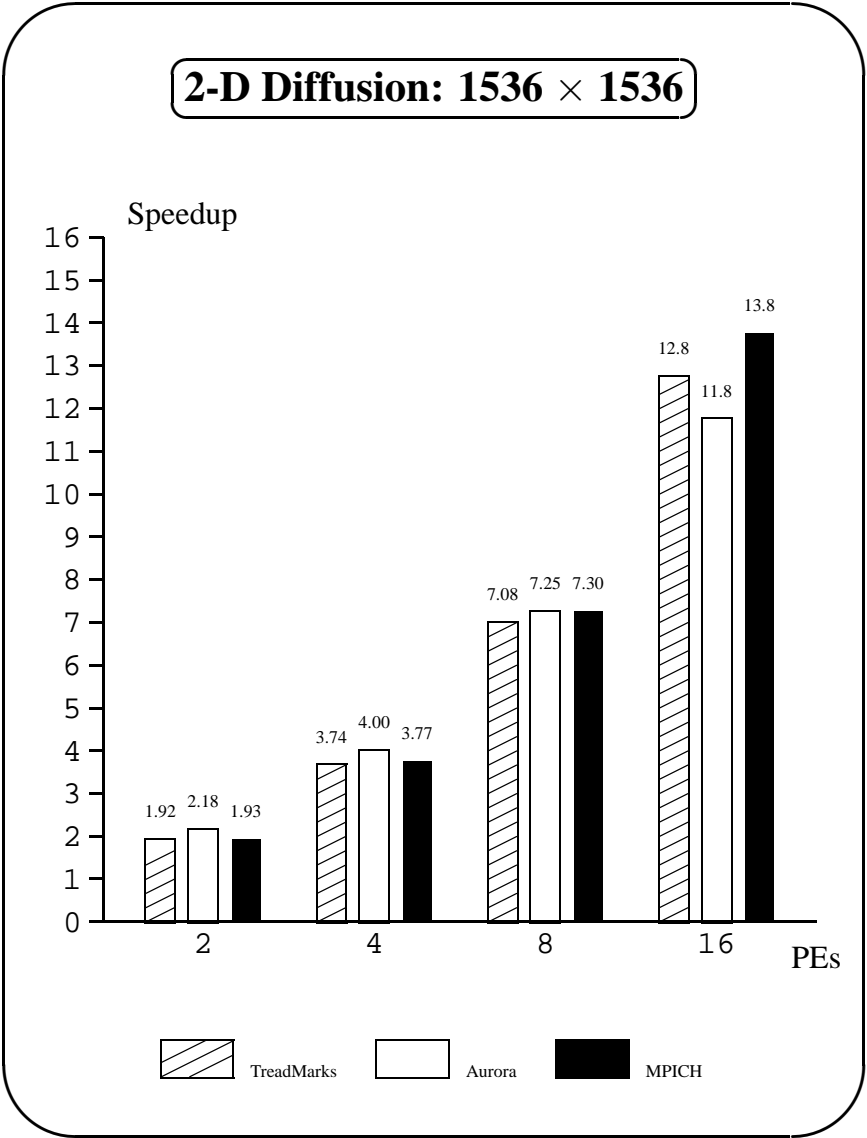
```
{
    NewBehaviour( Sample, GVReleaseC, int );
    for( i = ... )
        Sample[ i ] = ...           // Produce
}
```

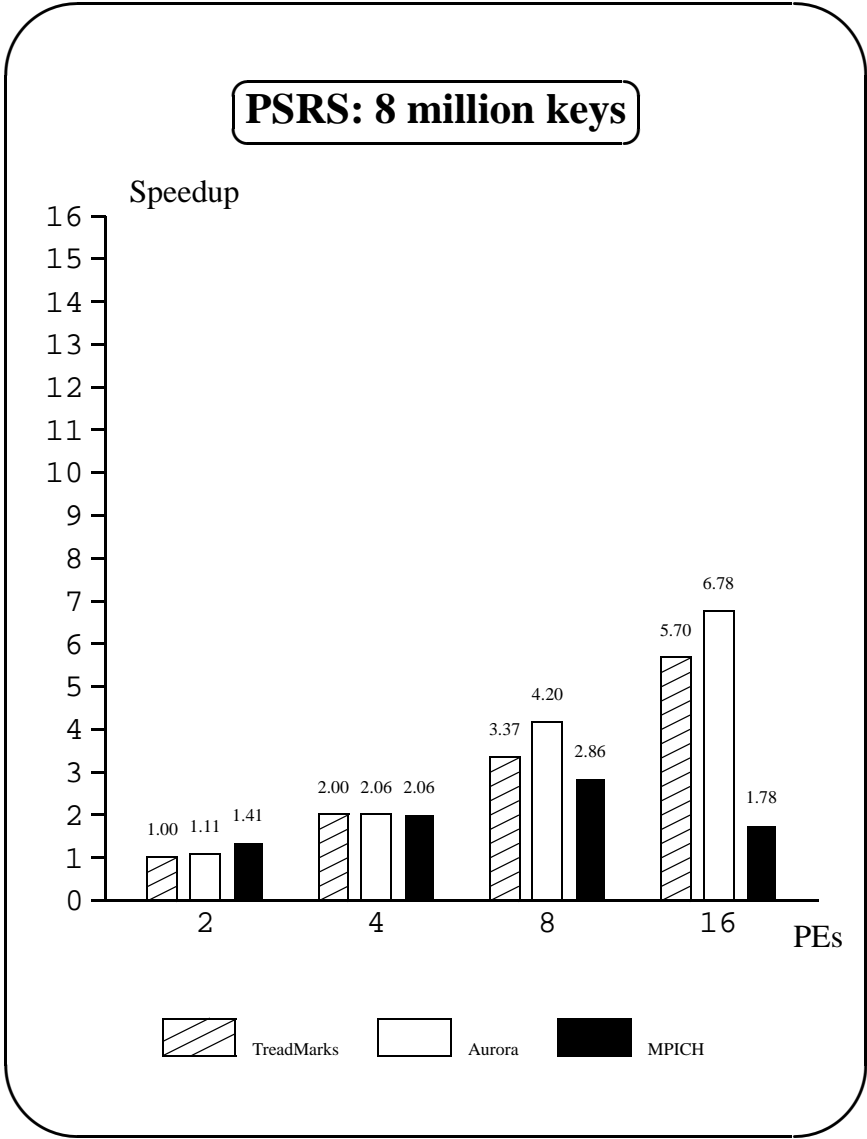
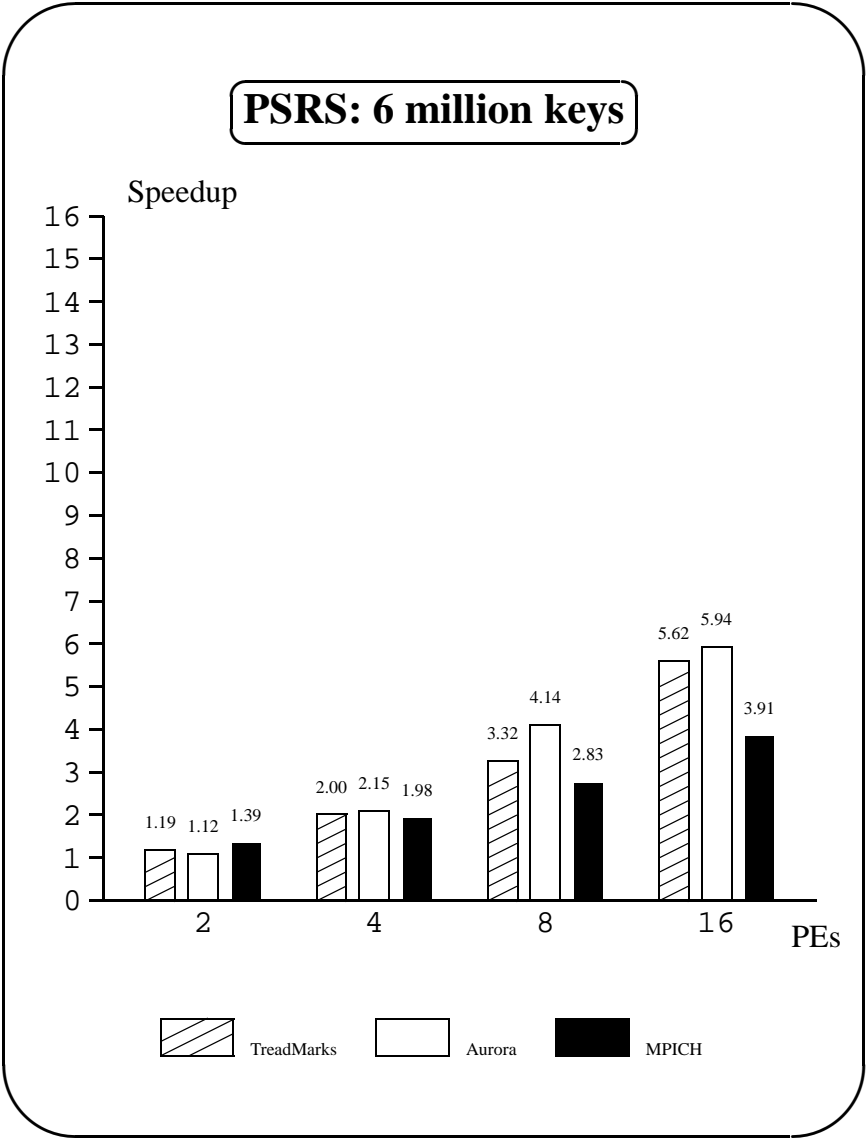


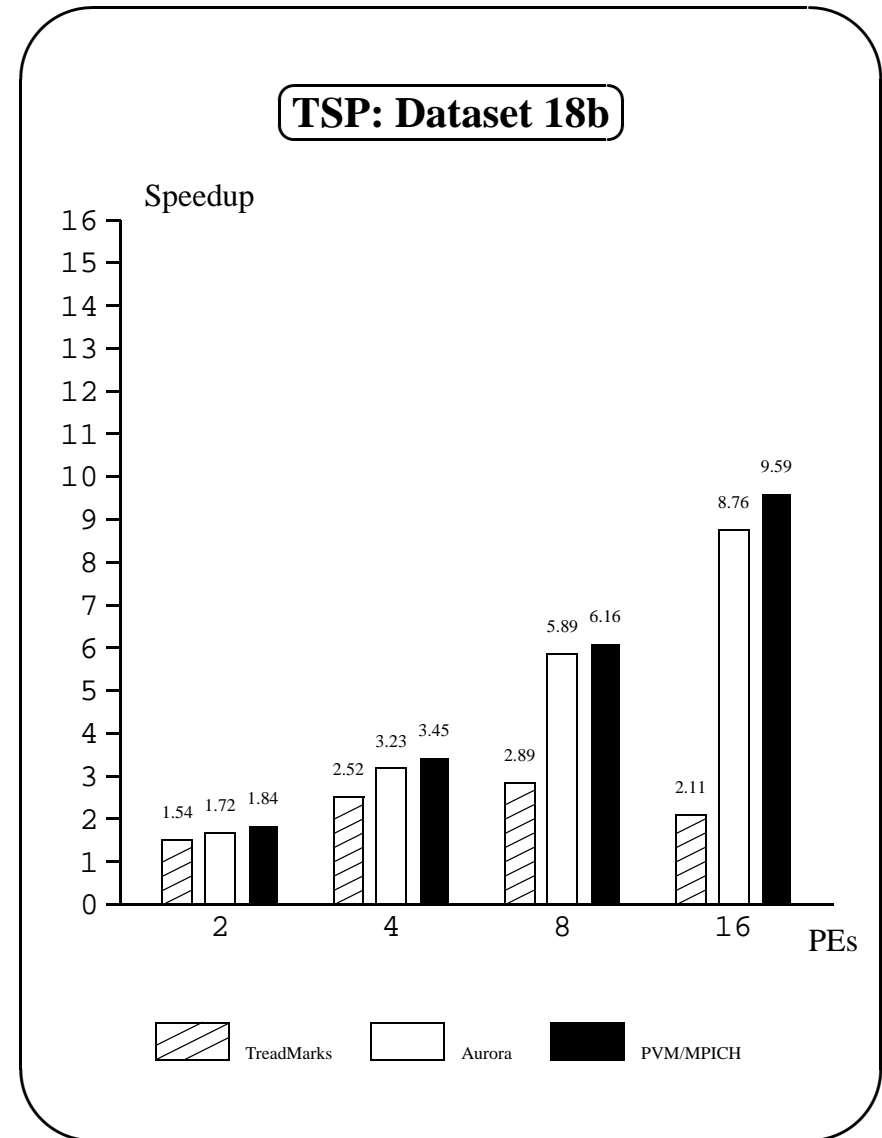
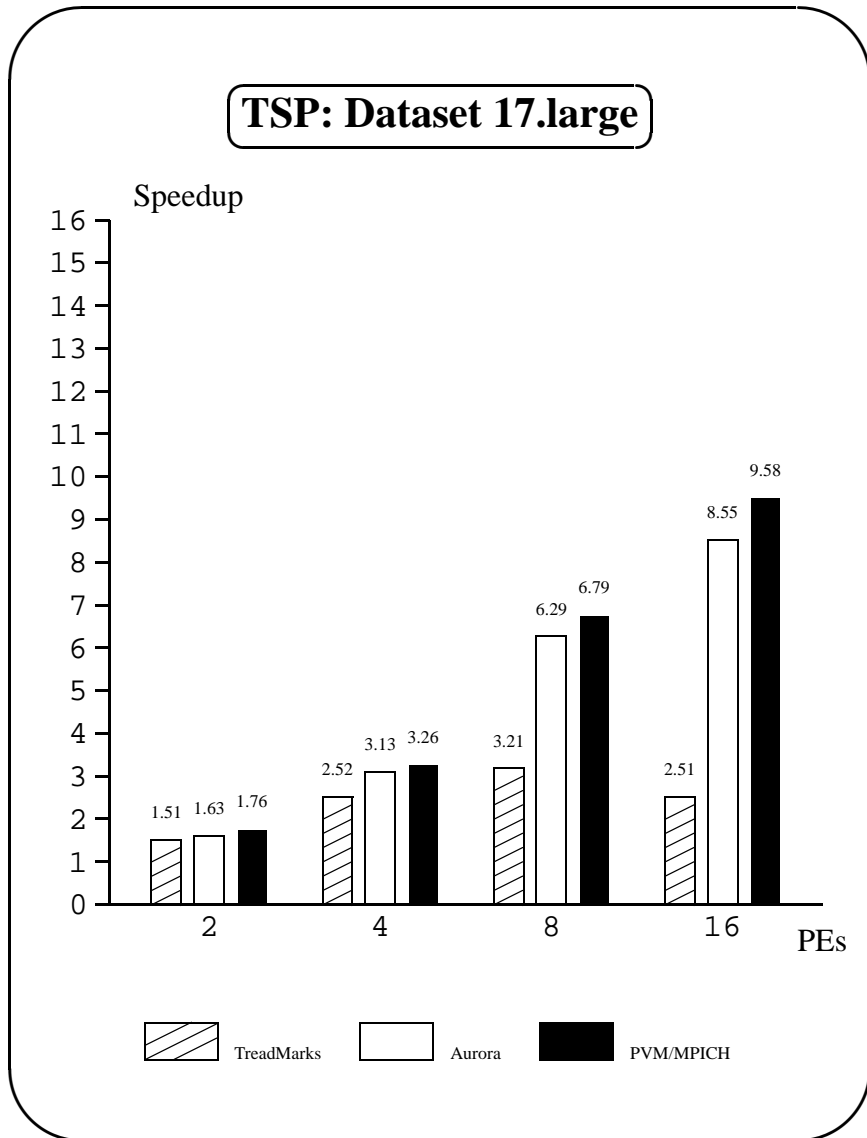
Phase 2: Node 0 is the *consumer* and uses:

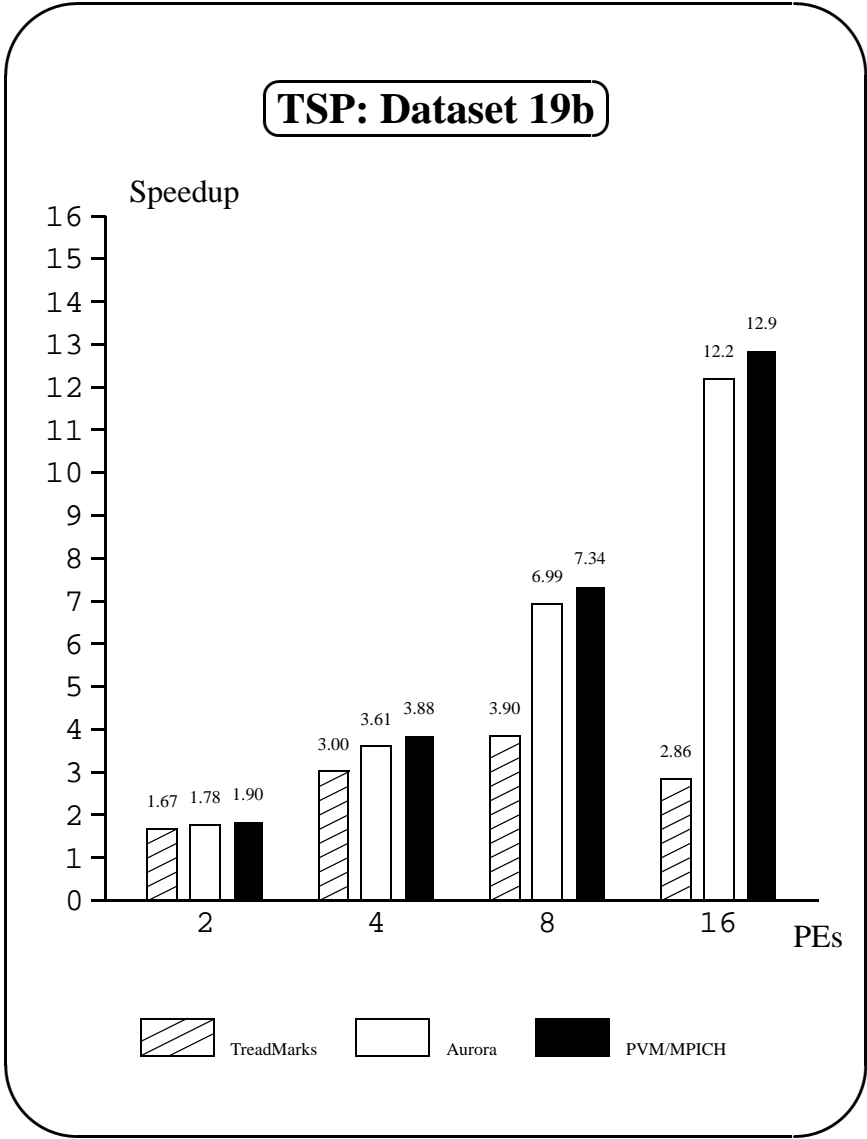
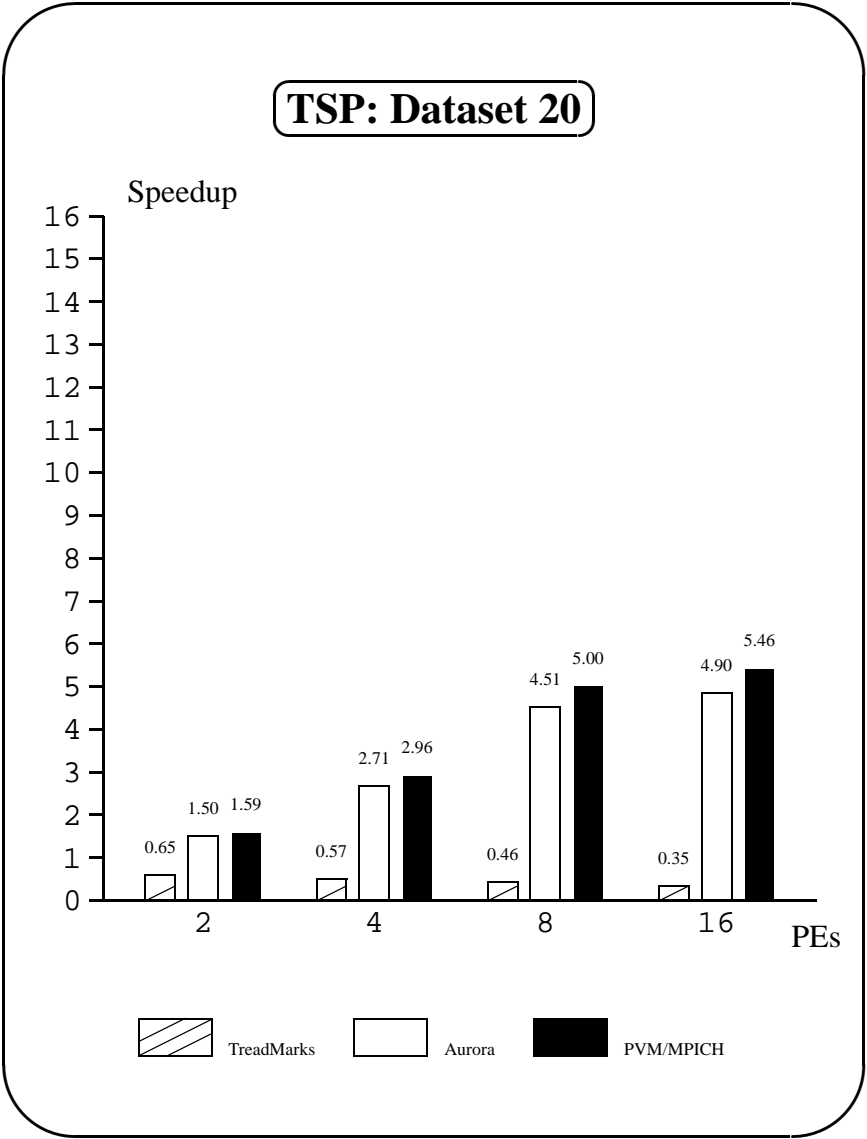
```
MASTER
{
    NewBehaviour( Sample, GVOwnerComputes, int );
    quicksort( Sample, 0, n - 1 );    // Consume
}
```











Summary of Performance

1. **TreadMarks** achieves good performance for regular problems with good locality of reference
2. **MPICH** achieves high performance, but has scalability problems with large data exchanges and large numbers of processors
3. **Aurora** generally comparable to or better than MPICH; usually faster than TreadMarks.
 - Significantly outperforms TreadMarks on TSP
 - Significantly outperforms MPICH on matrix multiplication and PSRS