Memory affinity in sparse matrix-vector multiplications on multicore NUMA architectures

Avinash Srinivasa
Masha Sosonkina
Ames Laboratory and Iowa State University

Work supported by National Science Foundation

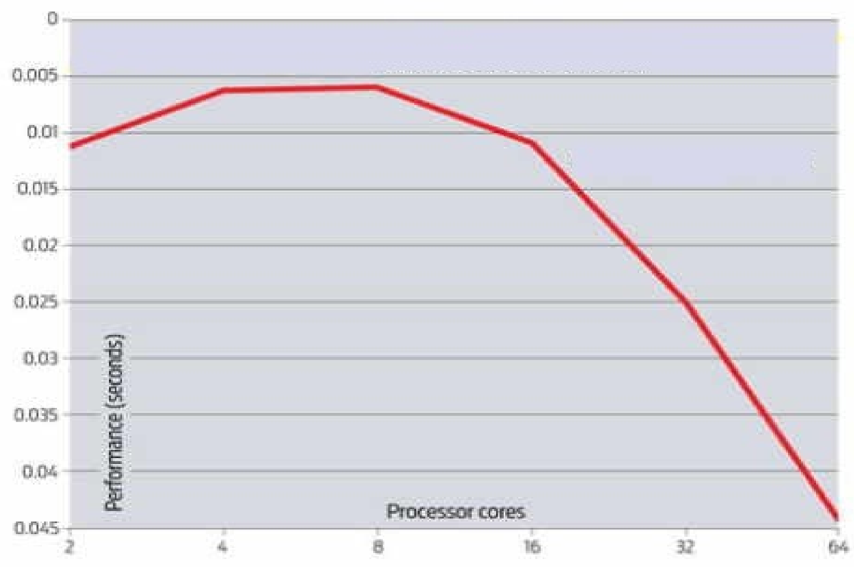
Sparse Days, Toulouse, Sept. 7 2011

Outline

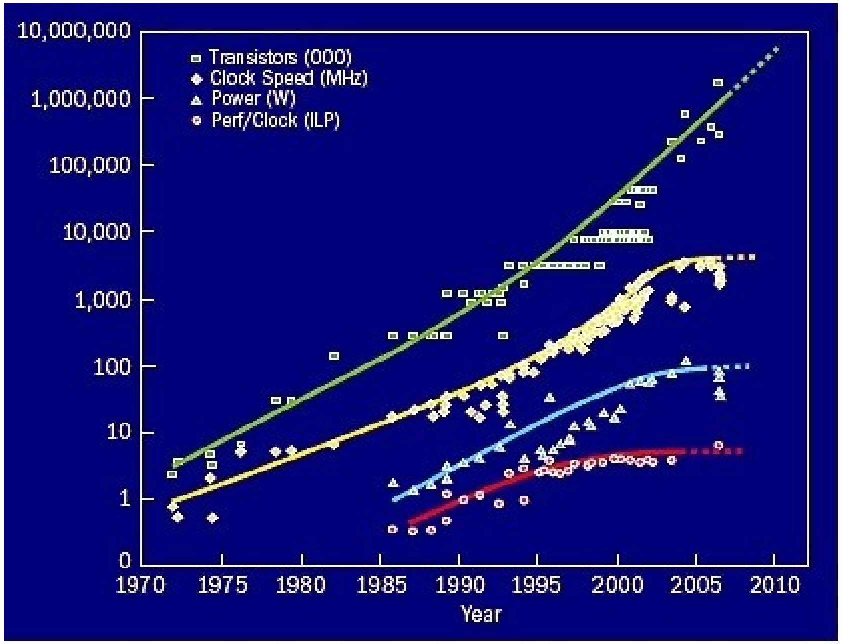
- Motivation: need for memory efficiency
- NUMA overview
- Existing memory affinity policy
- Proposed memory affinity policy
- Test applications
- Performance comparisons

Motivation

- Increasing core counts to with Moore’s law may cause
  - increased main memory contention;
  - reduced memory bandwidth and scalability applications;
- Introduction of NUMA (Non Uniform Memory Access) architectures – as a workaround to this problem.
  - characterized by multiple physical memory banks.
  - Memory affinity – placement of data in physical memory banks
  - needs to be efficient in order to accommodate NUMA architectures.



Variation of performance with core numbers for conventional architectures (**source:** Sandia National Labs)



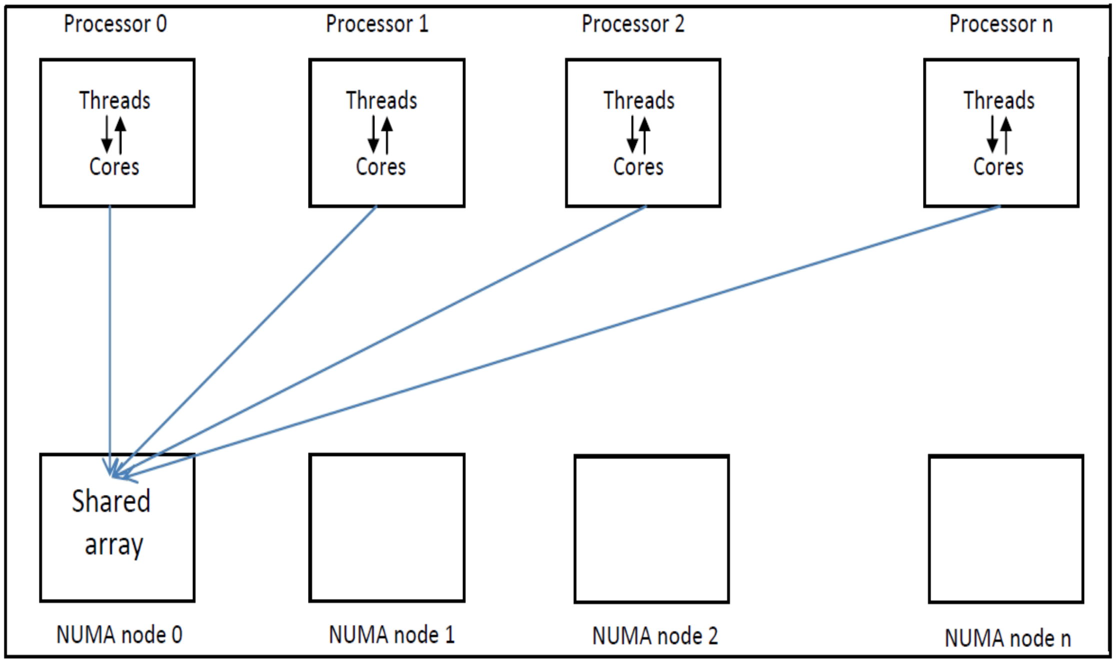
Impact of Moore's law (**source:** SCIDAC review)

NUMA overview

- NUMA – *Non Uniform Memory Access* – is becoming increasingly popular in the multicore domain.
- Characterized by *multiple physical memory banks* (NUMA nodes).
  - each provides the fastest access to a set of cores.
- Very effective for threads operating on *private data* due to improved memory bandwidth for large thread counts.
- *Shared data* among multiple threads.
  - remote access latencies and bandwidth contention.
  - calls for intelligent data distribution and placement (memory affinity) strategies.

Existing data placement policy

- Default memory affinity policy in Linux is *first touch.*
  - places memory on the NUMA node of the thread/process which initializes (touches) it first.
  - causes all threads which share a piece of memory to converge to the NUMA node which contains it.
  - results in remote access latencies and bandwidth contention.



Thread data access pattern with *first touch* policy

Proposed design/policy

(1) Identify all the shared arrays.
(2) Identify the shared arrays with *deterministic access* pattern:

- each thread exclusively accesses a certain contiguous chunk of the array.

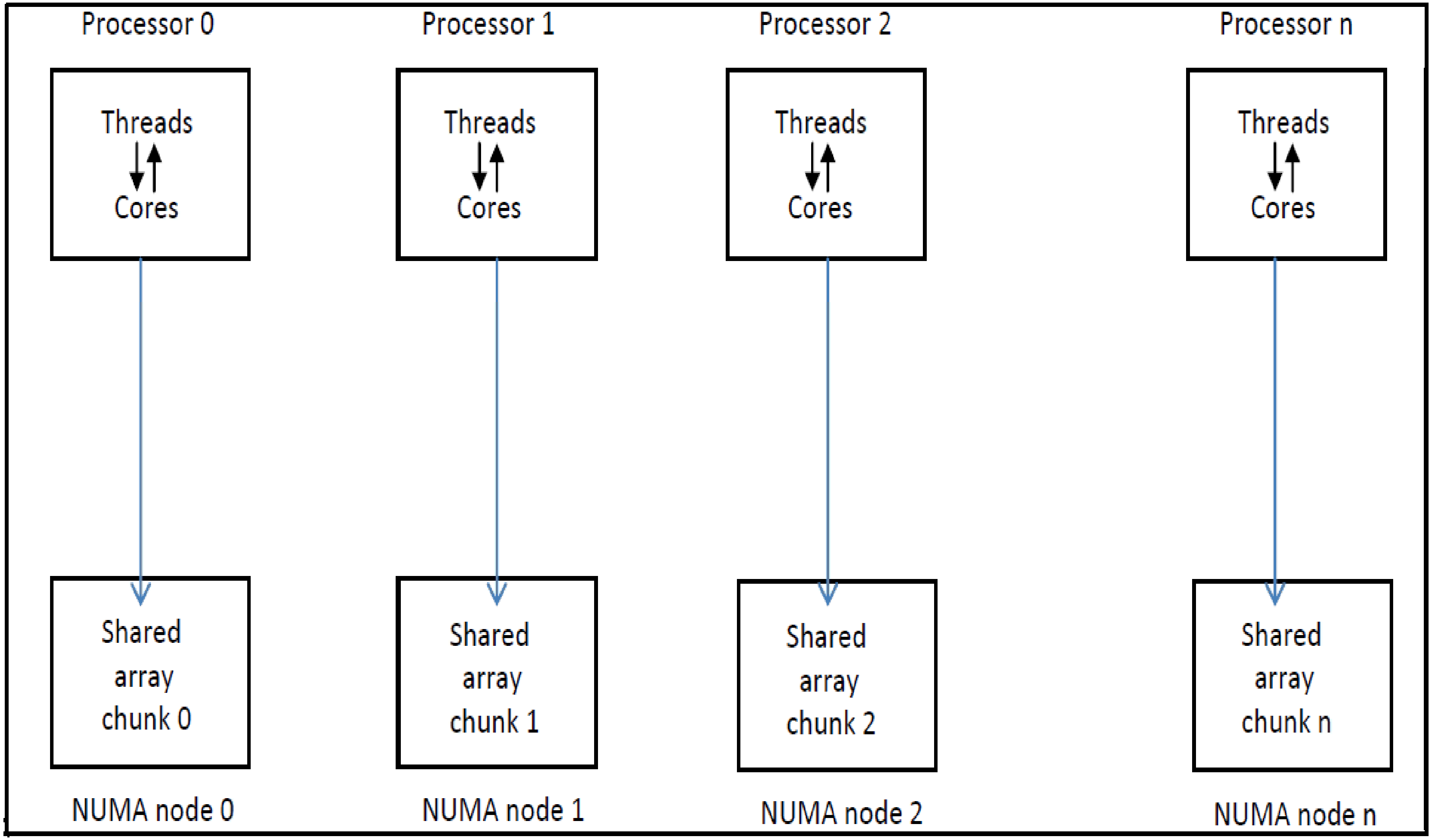
(3) Determine the chunks accessed by each thread and pin them to a specific memory bank.
(4) Identify the shared arrays with *non-deterministic access*:

- each thread has no exclusive access to a chunk → spread among banks.

ASSUMPTION

- *Static even distribution* of chunks among threads.
  - Static: work-sharing pattern is known in advance.
  - Even: each thread gets almost the same chunk size.
- Reason: loop iterations for each thread must be known apriori.
  - to place in memory the section of a shared array accessed by each thread before its execution.

Proposed Data Access Pattern



Thread data access pattern with proposed policy

Sparse Matrix Representation

- Sparse matrices – stored in CSC (Compressed Sparse Column) or CSR (Compressed Sparse Row) formats.
- CSC stored in column order and CSR in row order.
- Given, Matrix A = $\begin{matrix} 1 & 2 & 0 \\ 0 & 3 & 0 \\ 4 & 0 & 5 \end{matrix}$

	CSC		CSR
a( )	1 4 2 3 5	← non-zero elements →	1 2 3 4 5
ja( )	1 3 1 2 3	← row/column indices →	1 2 2 1 3
ia( )	1 3 5 6	← column/row pointers →	1 3 4 6

- CSC – Transpose of CSR and vice versa
- Matrix vector multiplication (MATVEC) – loop over the rows (CSR) or over the columns (CSC) – parallelized using a standard such as OpenMP.
- Matrix and vector arrays – shared between threads.

MATVEC pseudo-code

- MATVEC loop for CSC

```
DO i = 1, lastcol                    → Loop to be parallelized
    DO k = ia(i), ia(i+1)-1
      y(ja(k)) = y(ja(k)) + x(i)*a(k)   → Protected by critical section
    ENDDO
ENDDO
```

- MATVEC loop for CSR

```
DO i = 1, lastrow                    → Loop to be parallelized
    DO k = ia(i), ia(i+1)-1
      y(i) = y(i) + x(ja(k))*a(k)       → No need for critical section
    ENDDO
ENDDO
```

x( ) – input vector
y( ) – output vector

Shared-array affinity for MATVEC

- Two basic memory affinity policies provided by the Linux NUMA API, *bind and interleave*.
  - Bind: places (binds) data to a memory bank or set of banks.
  - Interleave: interleaves data on a page by page basis over the memory banks.
- Bind and Interleave are too generic to be applied across the entire application .
  - Solution: to apply them selectively according to the tread access patterns.
- Shared arrays of MATVEC distributed based on the access pattern of the threads.
  - Deterministic – bind sections to the appropriate NUMA bank.
  - Non-deterministic – interleave over all the NUMA banks.

CSC

Array	Access	Policy	Operation
a( )	D	Bind	Read
ja( )	D	Bind	Read
ia( )	D	Bind	Read
x( )	D	Bind	Read
y( )	ND	Intrlv.	Write

CSR

Array	Access	Policy	Operation
a( )	D	Bind	Read
ja( )	D	Bind	Read
ia( )	D	Bind	Read
x( )	ND	Intrlv.	Read
y( )	D	Bind	Write

D – Deterministic ND – Non-deterministic

Policy Implementation Pseudocode

To find no. of rows/columns (chunk) accessed per NUMA node:

```
per_thread_dim = ceil(dimension/nthreads)
virtual_dim = per_thread_dim*nthreads
offset = virtual_dim – dimension

loop i = 1, (nthreads-offset)
   dim_per_thread(i) = per_thread_dim
end loop

loop i = (nthreads-offset+1),  nthreads
   dim_per_thread(i) = per_thread_dim -1
end loop

loop j = 1, num_nodes
    loop i = num_cores*(j-1)+1, num_cores*j
        dim_per_node(j) += dim_per_thread(i)
   end loop
end loop
```

Here,
dimension – no. of rows/columns in the matrix
nthreads – no. of threads
dim_per_thread(i) – no. of rows/columns accessed by thread *i*
dim_per_node(i) – no. of rows/columns accessed by NUMA node *i*
num_nodes – no. of NUMA nodes
num_cores – no. of cores per NUMA node

Note:
1) This algorithm finds the most even distribution of chunks between the threads.
2) Chunk represents columns in the case of CSC and rows in the case of CSR.
3) Assumes no. of threads equals to no. of cores.

Test Applications: CG

- Kernel from the NAS parallel benchmark suite.
- Solves linear systems of equations with conjugate gradient method within an eigenvalue computation.
- Works on a large, sparse, symmetric matrix.
  - stored in CSR format.
- Employs sparse MATVECs (parallelized with OpenMP).
  - OpenMP C implementation is provided by the OMNI compiler group used for this work.
- Problem used:
  - Class C, matrix size 150,000.
  - single MPI process.

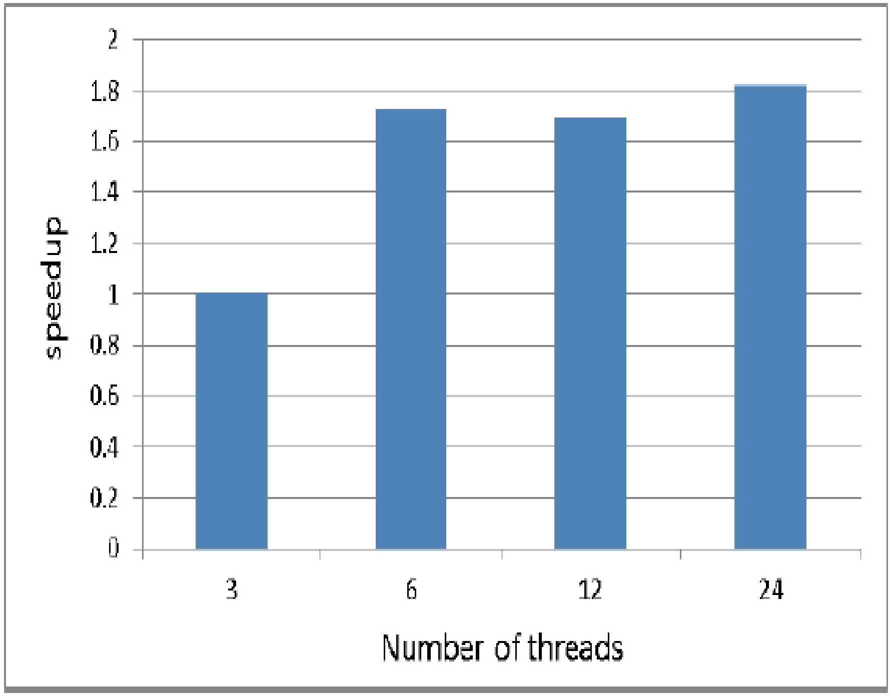
Test Applications: MFDn

- Computes ab initio nuclear structure using Lanczos algorithm.
- Large, sparse, symmetric nuclear Hamiltonian matrix is generated and stored in the CSC format.
- Diagonalized iteratively to obtain low lying eigenvalue and eigenvectors.
- Uses hybrid MPI/OpenMP
  - each MPI process spawns multiple OpenMP threads.
- Employs sparse MATVECs (parallelized with OpenMP).
- Symmetric matrix in MFDn -> only lower triangular portion stored:
  - MATVEC operation requires transpose matrix-vector multiplication.
- Problem used:
  - Carbon-12, $N_{max} = 4$ (N_{max} is maximum number of harmonic oscillator quanta).
  - 6 MPI processes (single process per compute node).

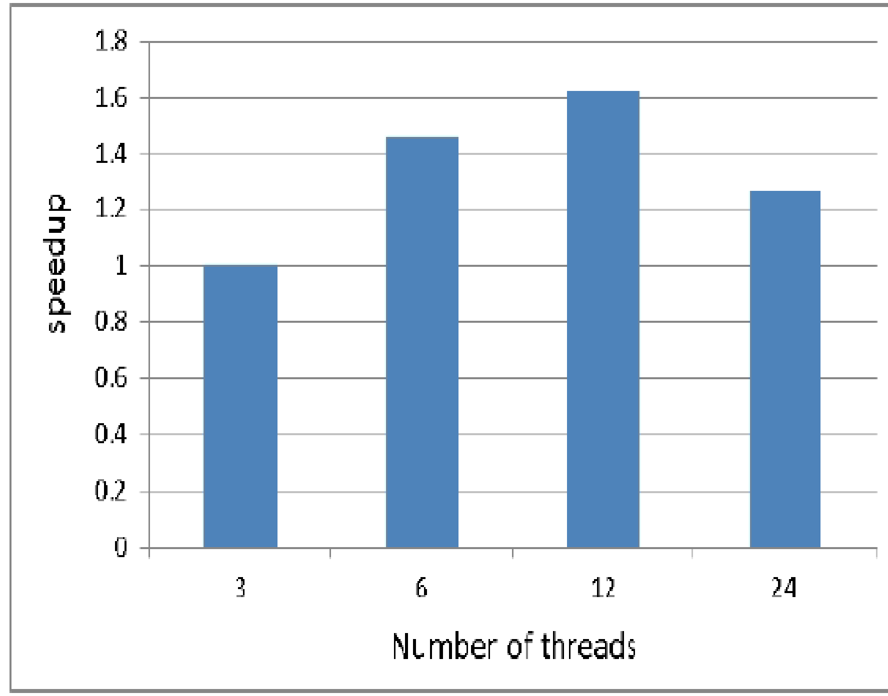
Performance Comparisons

- Testbed: Hopper supercomputer at NERSC.
  - 6,384 compute nodes.
  - Each node with 2 twelve-core AMD ‘MagnyCours’ processors (24 cores) and 32 GB of RAM.
  - NUMA architecture with 4 memory banks – each of 8 GB associated with a set of 6 cores.
- All speedups are with respect to wall clock time:
  - on a single process (running on one compute node).
  - for the MATVEC only.

Scaling with first-touch policy



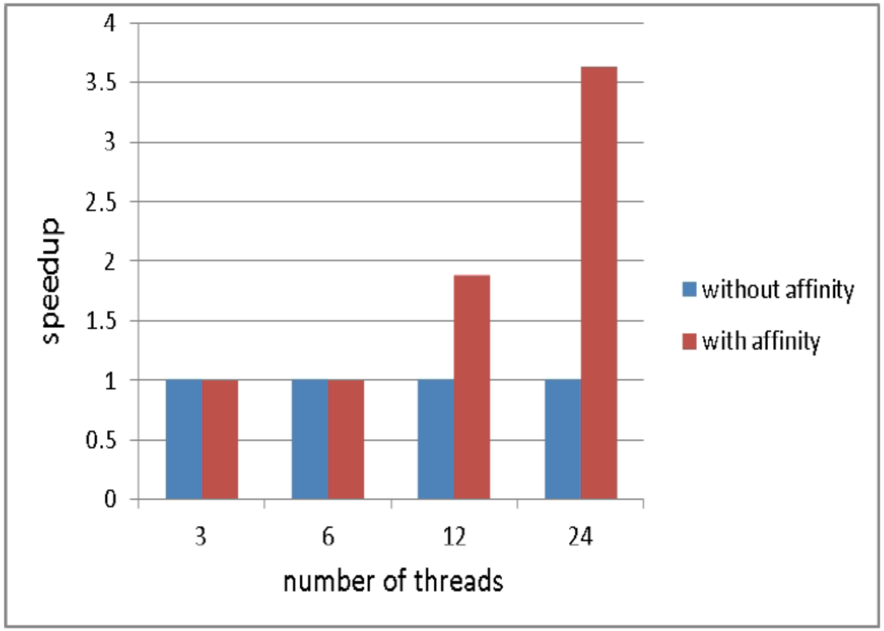
CG scaling without strategy



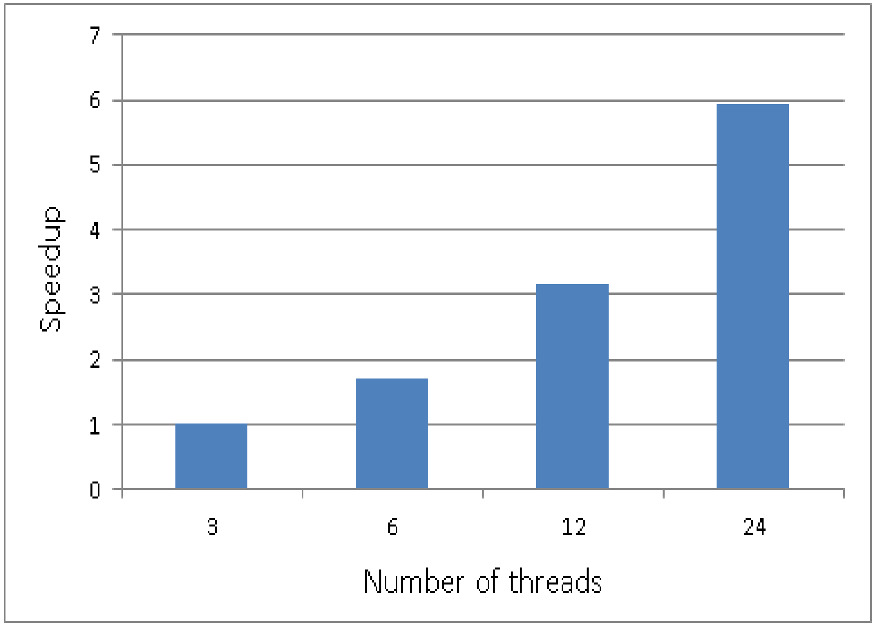
MFDn scaling without strategy

- Scaling suffers in moving beyond 6 threads because of NUMA effects.

CG with proposed policy



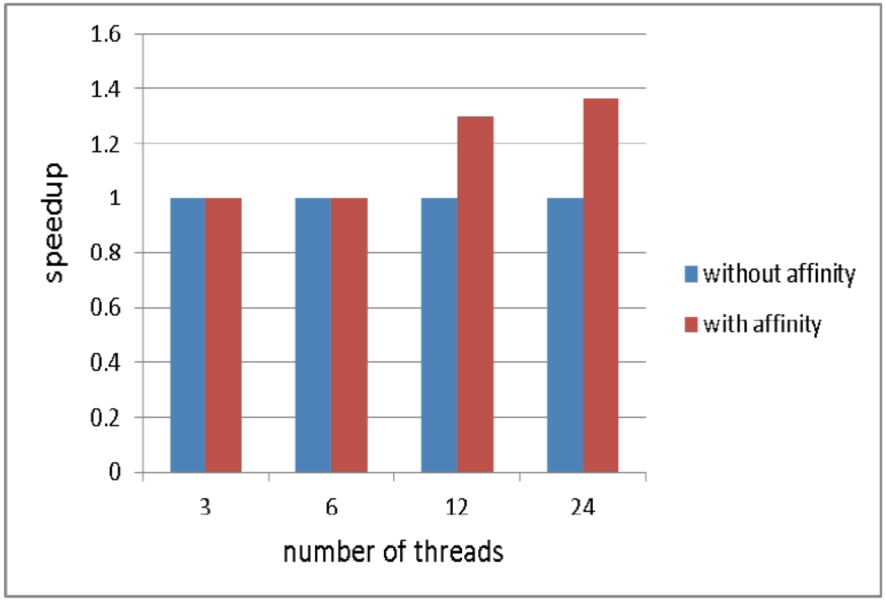
CG speedups



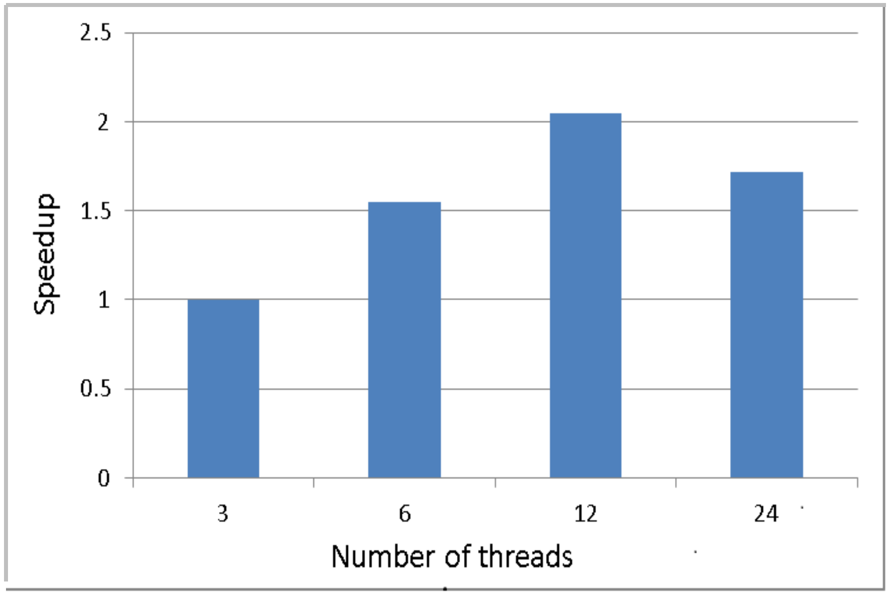
CG scaling with strategy

- CG scaling is shown to be almost ideal after applying the proposed policy – result of elimination of NUMA effects.

MFDn results with proposed policy



MFDn speedups

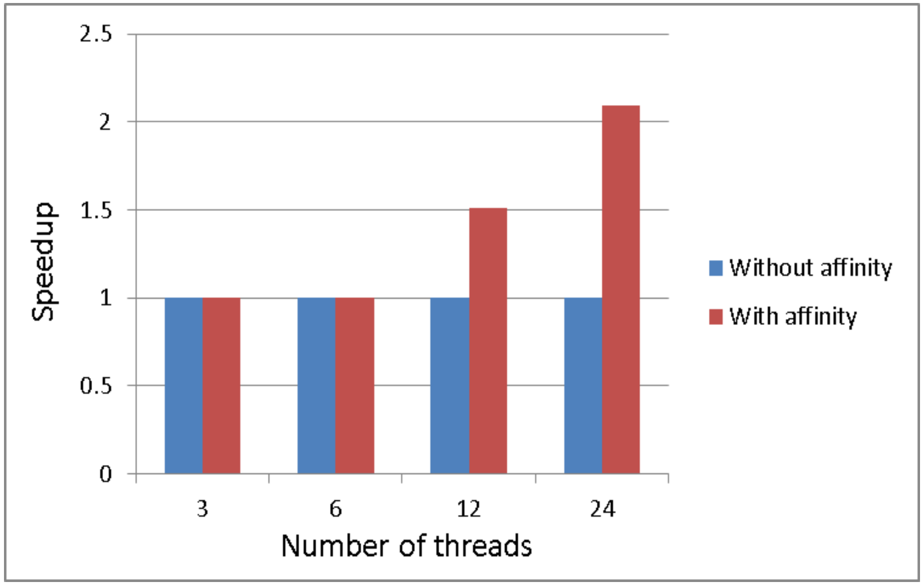


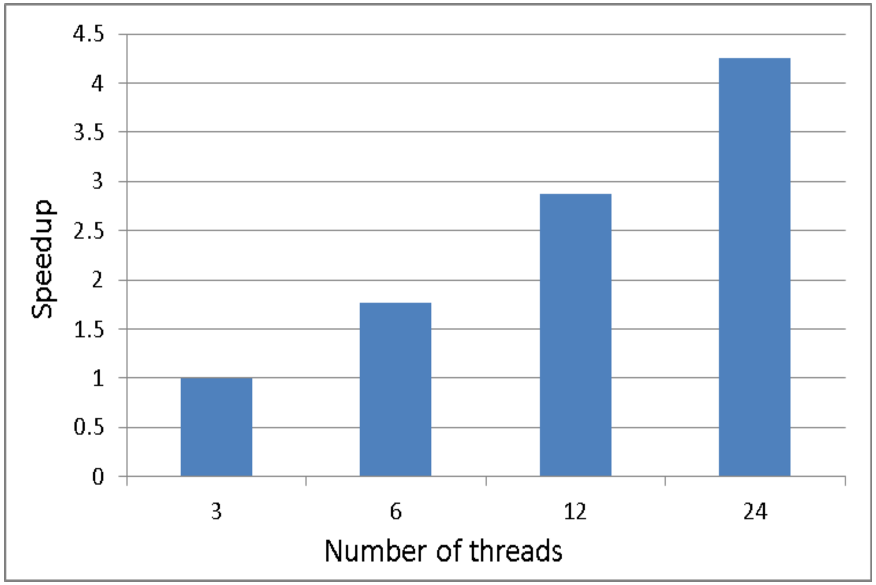
MFDn scaling with strategy

- Presence of critical section – irrespective of storage format used (CSC or CSR).
  - result of performing transpose MATVEC as well.
  - causes poor scaling at larger thread counts.

MFDn modified MATVEC

- Crude workaround: Use a combination of CSC and CSR with the proposed strategy.
- Doubles the memory requirement may not be practical for large problem sizes.





- Much improved speedups and consistent scaling.

Conclusions

- A memory affinity policy for efficient data placement is devised and implemented.
  - data placement for applications employing sparse matrix-vector multiplications (MATVECs).
  - designed for multicore NUMA architectures characterized by split physical memory banks.
- NUMA effects such as remote access latencies and bandwidth contention are eliminated for large thread/core counts.
  - speedup of up to 3 times observed over the default first-touch Linux policy.
  - consistent scaling among small to large thread counts.
- Generic nature of policy.
  - can be applied to any application employing sparse MATVECs.
  - can be extended to other parallel computations which show similar access patterns.