Optimizing Collective Communication on Multicores

Rajesh Nishtala[1] Katherine Yelick[1]

[1]University of California, Berkeley

(2009)

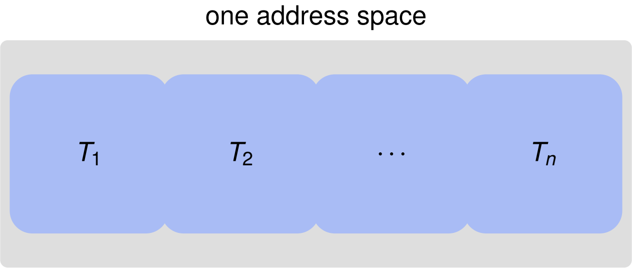
Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors

John M. Mellor-Crummey, Michael L.Scott
(1991)

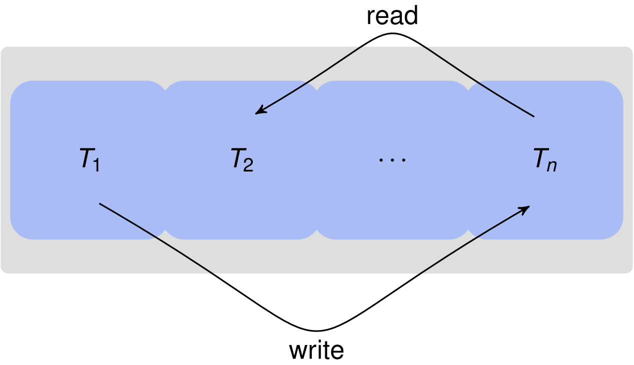
PGAS Languages

- Focus on Partitioned Global Address Space languages

Partitioned Addresspace



One Sided Communication

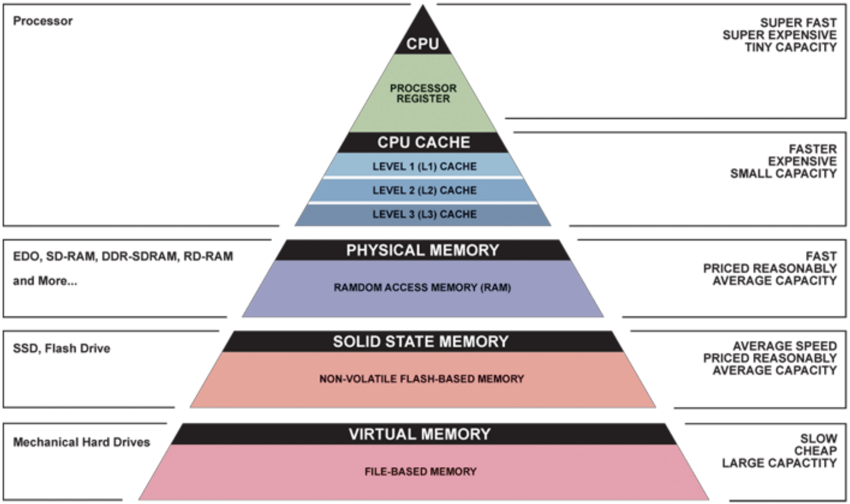


PGAS Languages

- UPC, Unified Parallel C
- CAF, Co-array Fortran
- Titanium, a Java dialect

Context

- The gap between processors and memory systems is still enormous



http://images.bit-tech.net/content_images/2007/11/the_secrets_of_pc_memory_part_1/hei.png

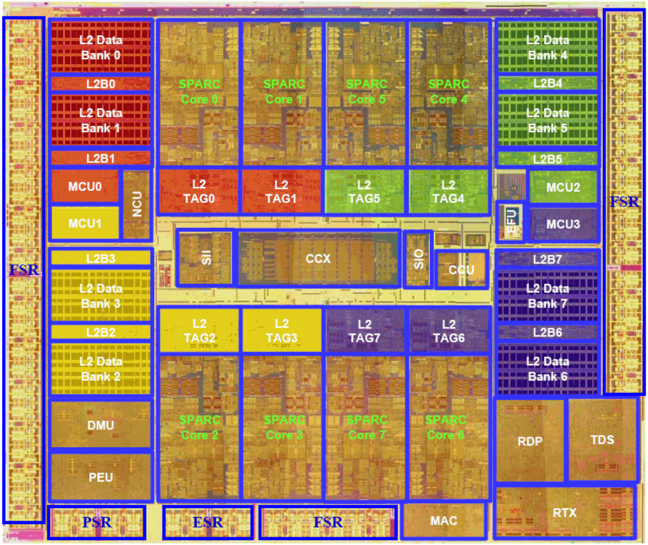
- Today: processors don't get faster, but we see more and more processors on a single chip

Processor	GHz	Cores (Threads)	Sockets
Intel Clovertown	2.66	8 (8)	2
AMD Barcelona	2.3	32 (32)	8
Sun Niagara 2	1.4	32 (256)	4

Table: Experimental Platforms

Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores

Sun Niagara 2

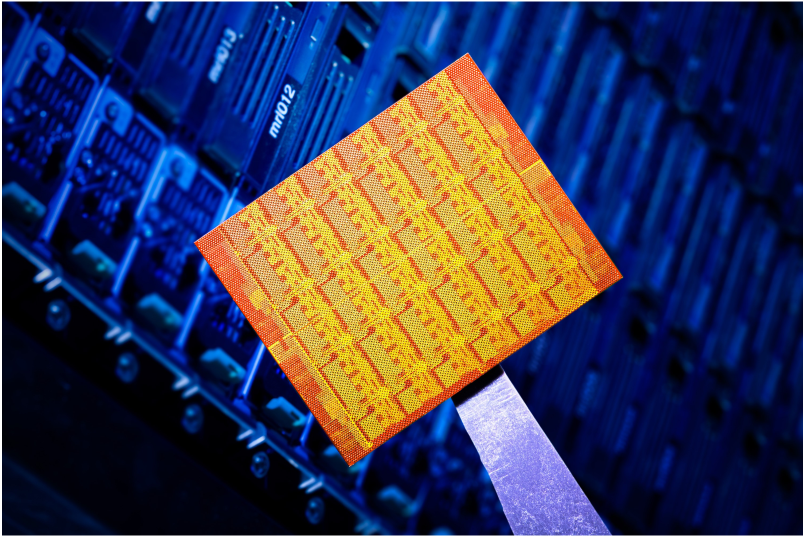


http://www.rz.rwth-aachen.de/aw/cms/rz/Themen/hochleistungsrechnen/ rechnersysteme/beschreibung_der_hpc_systeme/ultrasparc_t2/ rba/ultrasparc_t2_architectural_details/?lang=de

- The number of processors on a chip grows at an exponential pace

Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores

Intel Single-Chip Cloud Computer (48 Cores)

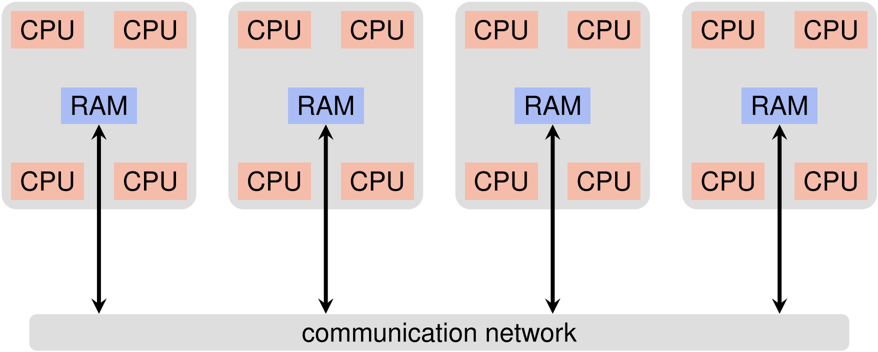


http://techresearch.intel.com/ProjectDetails.aspx?Id=1

- Communication in its most general form is the movement of data within cores, between cores or within memory systems

---

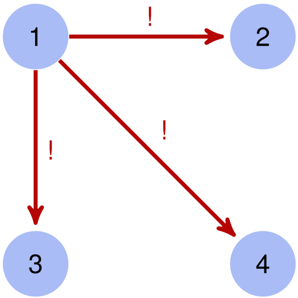
Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores


CPU
CPU
RAM
CPU
CPU
CPU
CPU
RAM
CPU
CPU
CPU
CPU
RAM
CPU
CPU
CPU
CPU
RAM
CPU
CPU
communication network

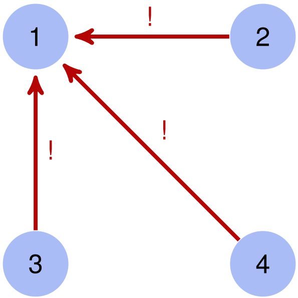
Collective Communication

- Communication-intensive problems often involve global communication

Broadcast



Gather



- These operations are thought of as collective communication operations

Example: Sum of Vector Elements

1	2	3	4	5	6	7	8	9	10

Example: Sum of Vector Elements

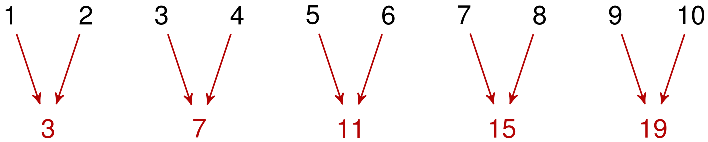
- Create workers

1 2 3 4 5 6 7 8 9 10

W_1 W_2 W_3 W_4 W_5

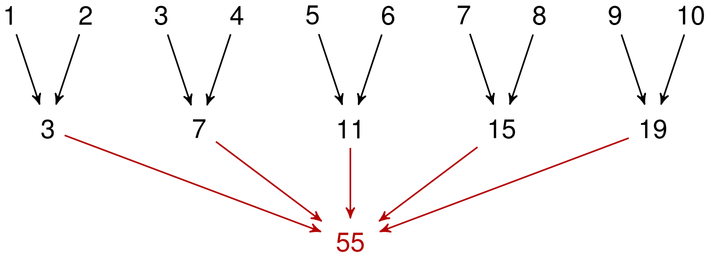
Example: Sum of Vector Elements

- Every worker sums up it’s part of the vector



Example: Sum of Vector Elements

- The main thread gathers the partial results and sums them up



Example: Sum of Vector Elements

Pseudocode (main thread):

```
double [] vector = read_vector();
Thread [] workers = spwan_workers();

start_workers(workers);
double result = calculate_result(workers);
```

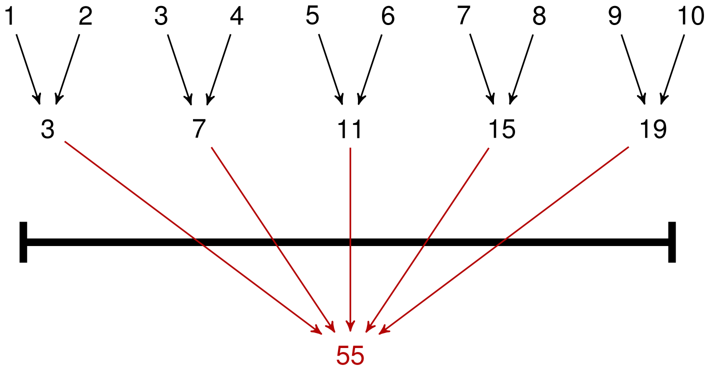
Example: Sum of Vector Elements

Pseudocode (main thread):

```
double [] vector = read_vector();
Thread [] workers = spwan_workers();

start_workers(workers);
wait_until_everything_finished(workers);
double result = calculate_result(workers);
```

Barrier

- Synchronization method for a group of threads
- A thread can only continue it's execution after every thread has called the barrier


1
2
3
4
5
6
7
8
9
10
3
7
11
15
19
55

Collective Communication Operation

"... group of threads works together to perform a global communication operation ..."

Reduce

- Divide a problem into smaller subproblems
- Every thread contributes it’s part to the solution
- Example: Calculate the smallest entry of a vector

Flat vs. Tree

- For communication among threads, different topologies can be used

Flat Topology

- Example: we have a reduce operation
- in the end the main thread W_{main} has to wait for every worker thread $W_1, \ldots, W_7$

W_{main} W_1 W_2 W_3 W_4 W_5 W_6 W_7

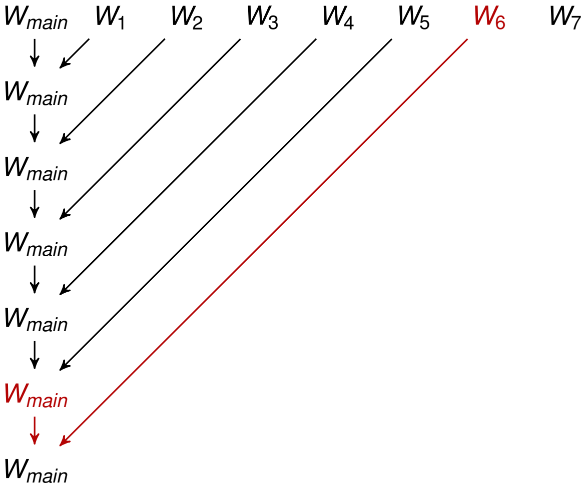
$$\begin{tikzcd}
W_{main} \arrow[d] & W_1 \arrow[dl] & W_2 & W_3 & W_4 & W_5 & W_6 & W_7 \\
W_{main} & & & & & & &
\end{tikzcd}$$

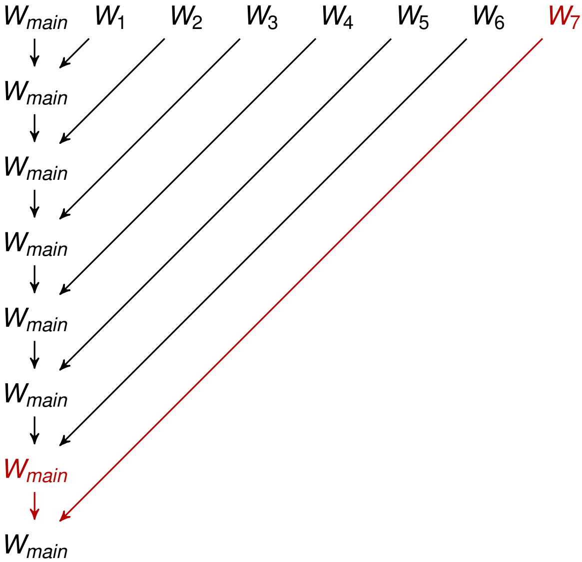
$$
\begin{tikzcd}
W_{main} \arrow[d] & W_1 \arrow[dl] & W_2 \arrow[ddll] & W_3 & W_4 & W_5 & W_6 & W_7 \\
W_{main} \arrow[d] & & & & & & & \\
W_{main} & & & & & & &
\end{tikzcd}
$$

$$
\begin{tikzcd}
W_{main} \arrow[d] & W_1 \arrow[dl] & W_2 \arrow[ddll] & W_3 \arrow[dddlll] & W_4 & W_5 & W_6 & W_7 \\
W_{main} \arrow[d] & & & & & & & \\
W_{main} \arrow[d] & & & & & & & \\
W_{main} & & & & & & &
\end{tikzcd}
$$

$$
\begin{tikzcd}
W_{main} \arrow[d] & W_1 \arrow[dl] & W_2 \arrow[ddll] & W_3 \arrow[dddlll] & \color{red}{W_4} \arrow[ddddllll, red] & W_5 & W_6 & W_7 \\
W_{main} \arrow[d] & & & & & & & \\
W_{main} \arrow[d] & & & & & & & \\
\color{red}{W_{main}} \arrow[d, red] & & & & & & & \\
W_{main} & & & & & & &
\end{tikzcd}
$$

$$
\begin{tikzcd}
W_{main} \arrow[d] & W_1 \arrow[dl] & W_2 \arrow[ddll] & W_3 \arrow[dddlll] & W_4 \arrow[ddddllll] & W_5 \arrow[dddddlllll] & W_6 & W_7 \\
W_{main} \arrow[d] \\
W_{main} \arrow[d] \\
W_{main} \arrow[d] \\
W_{main} \arrow[d] \\
W_{main}
\end{tikzcd}
$$


W_{main} W_1 W_2 W_3 W_4 W_5 W_6 W_7
W_{main}
W_{main}
W_{main}
W_{main}
W_{main}
W_{main}


W_{main} W_1 W_2 W_3 W_4 W_5 W_6 W_7
W_{main}
W_{main}
W_{main}
W_{main}
W_{main}
W_{main}
W_{main}

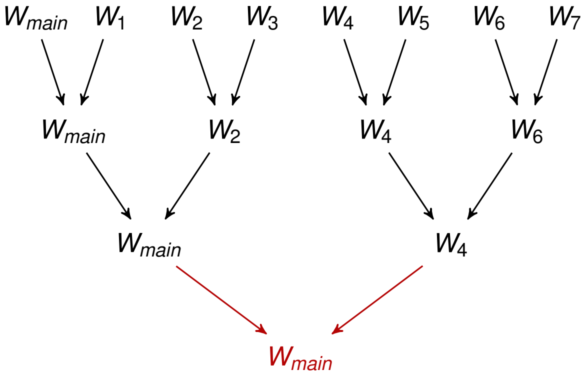
Tree Topology

- Example: we have a reduce operation
- in the end the main thread W_{main} has to wait for every worker thread $W_1, \ldots, W_7$

$$W_{main} \quad W_1 \quad W_2 \quad W_3 \quad W_4 \quad W_5 \quad W_6 \quad W_7$$

$$\begin{tikzcd}[column sep=small]
W_{main} \arrow[dr] & & W_1 \arrow[dl] & W_2 \arrow[dr] & & W_3 \arrow[dl] & W_4 \arrow[dr] & & W_5 \arrow[dl] & W_6 \arrow[dr] & & W_7 \arrow[dl] \\
& W_{main} & & & W_2 & & & W_4 & & & W_6 &
\end{tikzcd}$$

$$
\begin{tikzcd}
W_{main} \arrow[d] & W_1 \arrow[dl] & W_2 \arrow[d] & W_3 \arrow[dl] & W_4 \arrow[d] & W_5 \arrow[dl] & W_6 \arrow[d] & W_7 \arrow[dl] \\
W_{main} \arrow[d] & & W_2 \arrow[dll] & & W_4 \arrow[d] & & W_6 \arrow[dll] & \\
W_{main} & & & & W_4 & & &
\end{tikzcd}
$$


W_{main} W_1 W_2 W_3 W_4 W_5 W_6 W_7
W_{main} W_2 W_4 W_6
W_{main} W_4
W_{main}

Analysis

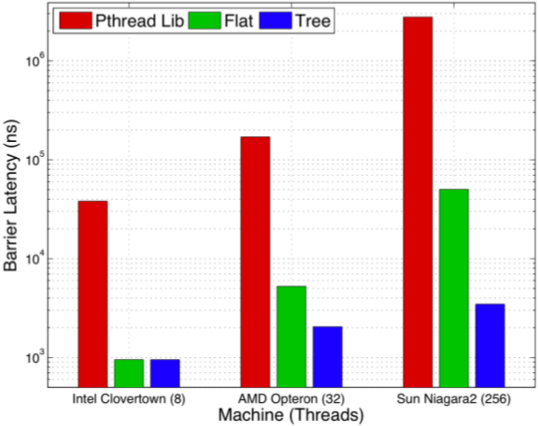


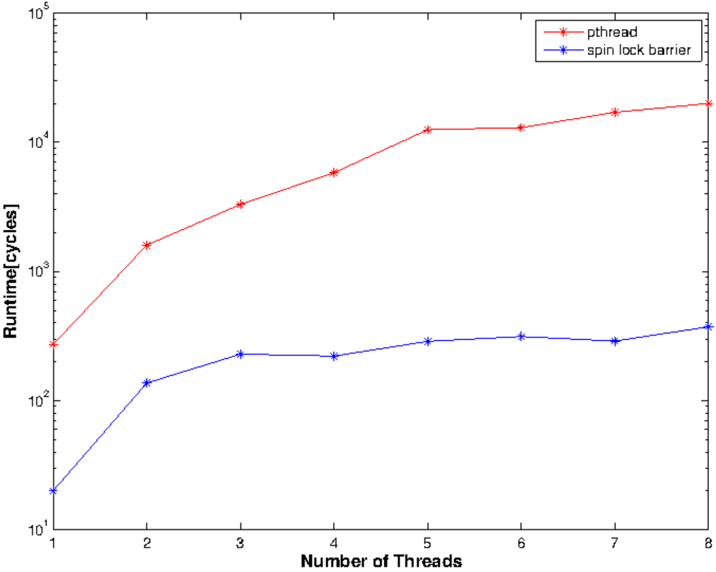
Figure: Barrier Performance

Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores

Barrier Implementation

```c
#define N 4

pthread_t threads[N];
volatile int ready[N];
volatile int go[N];

void barrier(int id) {

    if (id == 0) {
        //wait for each thread
        for (int i = 1; i < N; i++)
            while (ready[i] == 0);

        //reset the ready flags
        for (int i = 0; i < N; i++)
            ready[i] = 0;

        //signal each thread
        for (int i = 0; i < N; i++)
            go[i] = 1;
    }
    else {
        ready[id] = 1;
        //wait until thread is signalled
        while (go[id] == 0);
        go[id] = 0;
    }
}
```

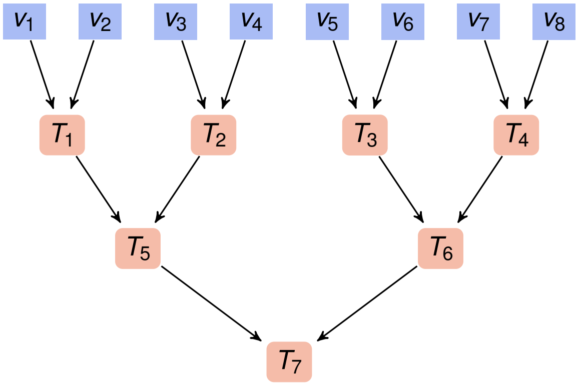
Experiment: Barrier Implementation



- Strict synchronization: Data movement can only start after all threads have entered the collective and must be completed before the first thread exits the collective

Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores

Strict Synchronization

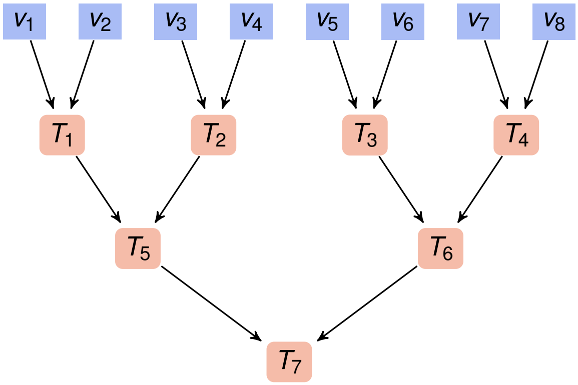


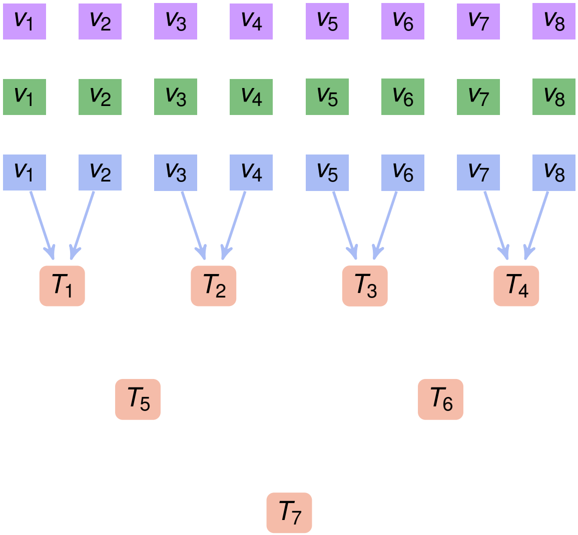
Loosening Synchronization Requirements

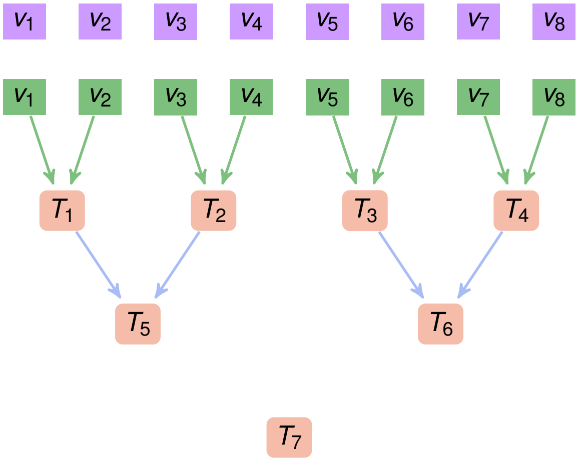
- Loose synchronization: Data movement can begin as soon as any thread has entered the collective and continue until the last thread leaves the collective

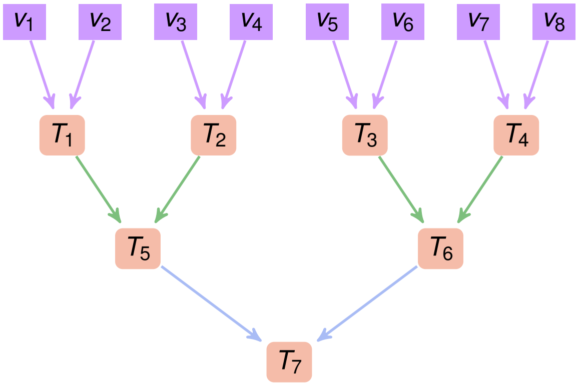
Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores

Loose Synchronization




v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
T_1 T_2 T_3 T_4
T_5 T_6
T_7


v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
T_1
T_2
T_3
T_4
T_5
T_6
T_7



v_1
v_2
v_3
v_4
v_5
v_6
v_7
v_8
T_1
T_2
T_3
T_4
T_5
T_6
T_7

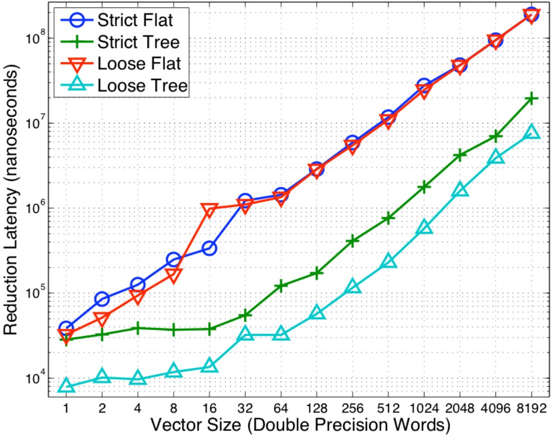


Figure 5: Optimal Algorithm Selection on Niagara2

(32 cores, 256 threads)

Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores

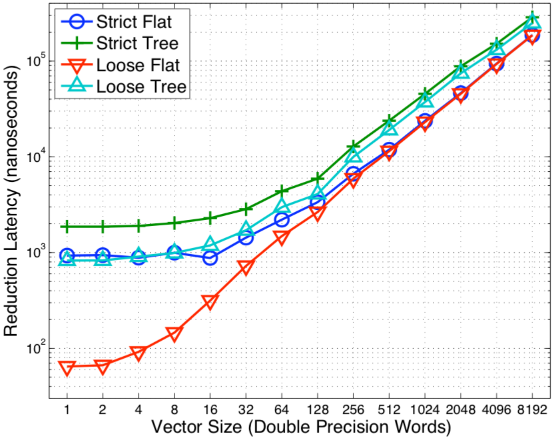


Figure 6: Optimal Algorithm Selection on Clovertown

(8 cores, 8 threads)

Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores

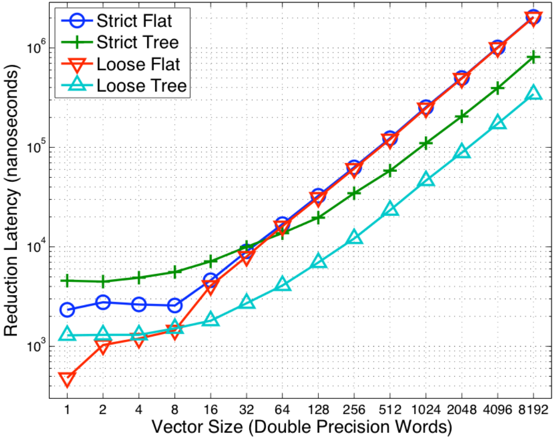


Figure 4: Optimal Algorithm Selection on Barcelona

(32 cores, 32 threads)

Nishtala, R., Yelick, K. Optimizing Collective Communication on Multicores

Summary

- Best strategy depends on the hardware and on the problem
- Using a library that can automatically adapt to a given situation can bring a great performance improvement, since hand tuning takes far too long

Words on the Paper

- Very high level
- Description of the problem without concrete solution
- No implementation
- Plots aren't always clear and precise