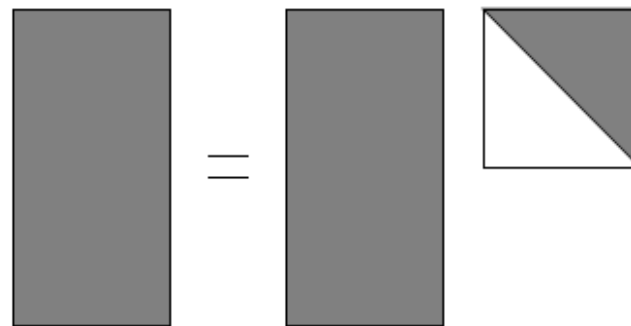


Parallelization of the Classic Gram-Schmidt QR-Factorization

Algorithm: Classical Gram-Schmidt

```
for j = 1:n
   $\tilde{\mathbf{v}}_j = \tilde{\mathbf{a}}_j$ 
  for i=1:(j-1)
     $r_{ij} = \tilde{\mathbf{q}}_i^* \tilde{\mathbf{a}}_j$ 
     $\tilde{\mathbf{v}}_j = \tilde{\mathbf{v}}_j - r_{ij} \tilde{\mathbf{q}}_i$ 
  endfor-i
   $r_{jj} = \|\tilde{\mathbf{v}}_j\|_2$ 
   $\tilde{\mathbf{q}}_j = \tilde{\mathbf{v}}_j / r_{jj}$ 
endfor-j
```

$$\mathbf{A} = \hat{\mathbf{Q}}\hat{\mathbf{R}}$$



Overview

- ▣ Introduction
- ▣ MATLAB vs. FORTRAN
- ▣ Parallelization with MPI
- ▣ Parallel Computation Results
- ▣ Conclusion

Introduction



“It is well established that the classical Gram-Schmidt process is one of the unstable ones. Consequently it is rarely used, except sometimes on parallel computers in situations where advantages in communication may outweigh the disadvantage of instability.”

- Trefethem-Bau pg. 66

MATLAB vs. FORTRAN

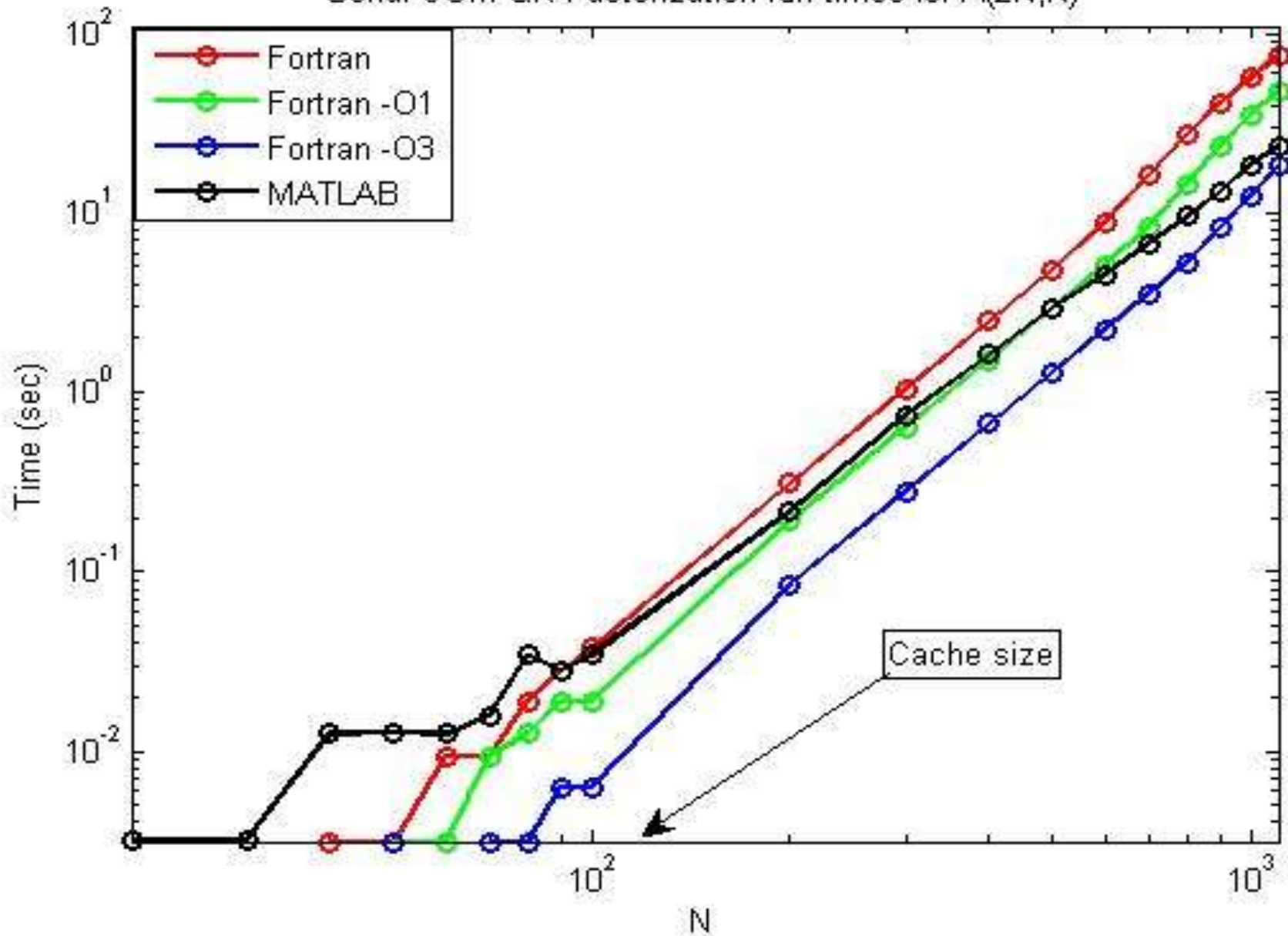
Pentium 4 2.0 Ghz with 512 KB cache, 512MB RDRAM running Windows XP
MATLAB Student Version 7 and g95 with cygwin

Each run is done 5 times and the times averaged.
In-between runs, memory is cleared.

<pre>call etime(elapsed,t1) do j = 1,N v_j = A(:,j) do i = 1,(j-1) R(i,j) = dot_product(Q(:,i),A(:,j)) v_j = v_j - (R(i,j)*Q(:,i)) enddo call norm(v_j,M,R(j,j)) Q(:,j) = v_j/R(j,j) enddo call etime(elapsed,t2)</pre>		<pre>t = clock; for j = 1:n v_j = A(:,j); for i = 1:(j-1) R(i,j) = Q(:,i)'*A(:,j); v_j = v_j - R(i,j)*Q(:,i); end R(j,j) = norm(v_j,2); Q(:,j) = v_j/R(j,j); end endtime = etime(clock, t);</pre>	
---	---	---	---

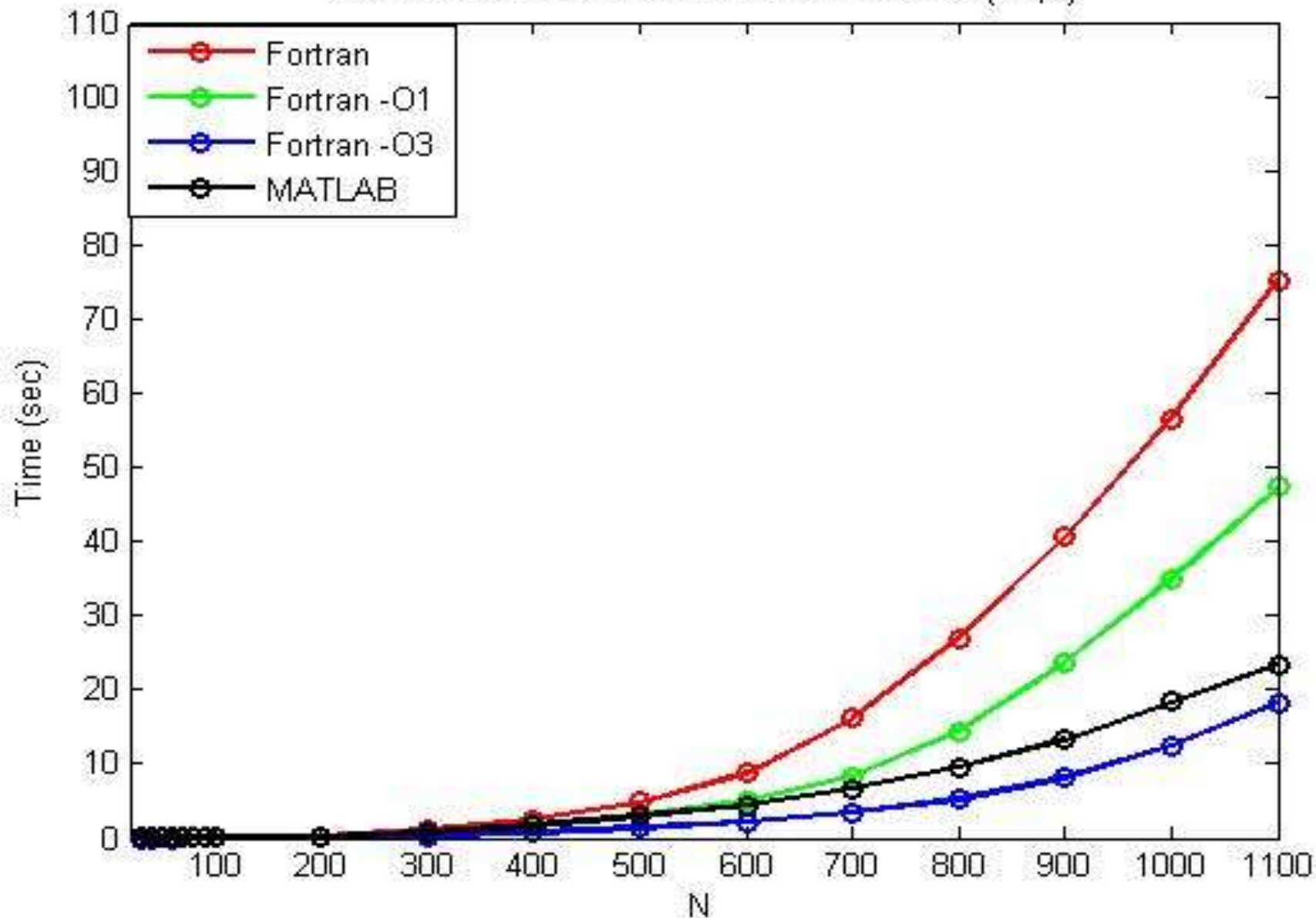
MATLAB vs. FORTRAN

Serial CGM QR-Factorization run times for $A(2N,N)$

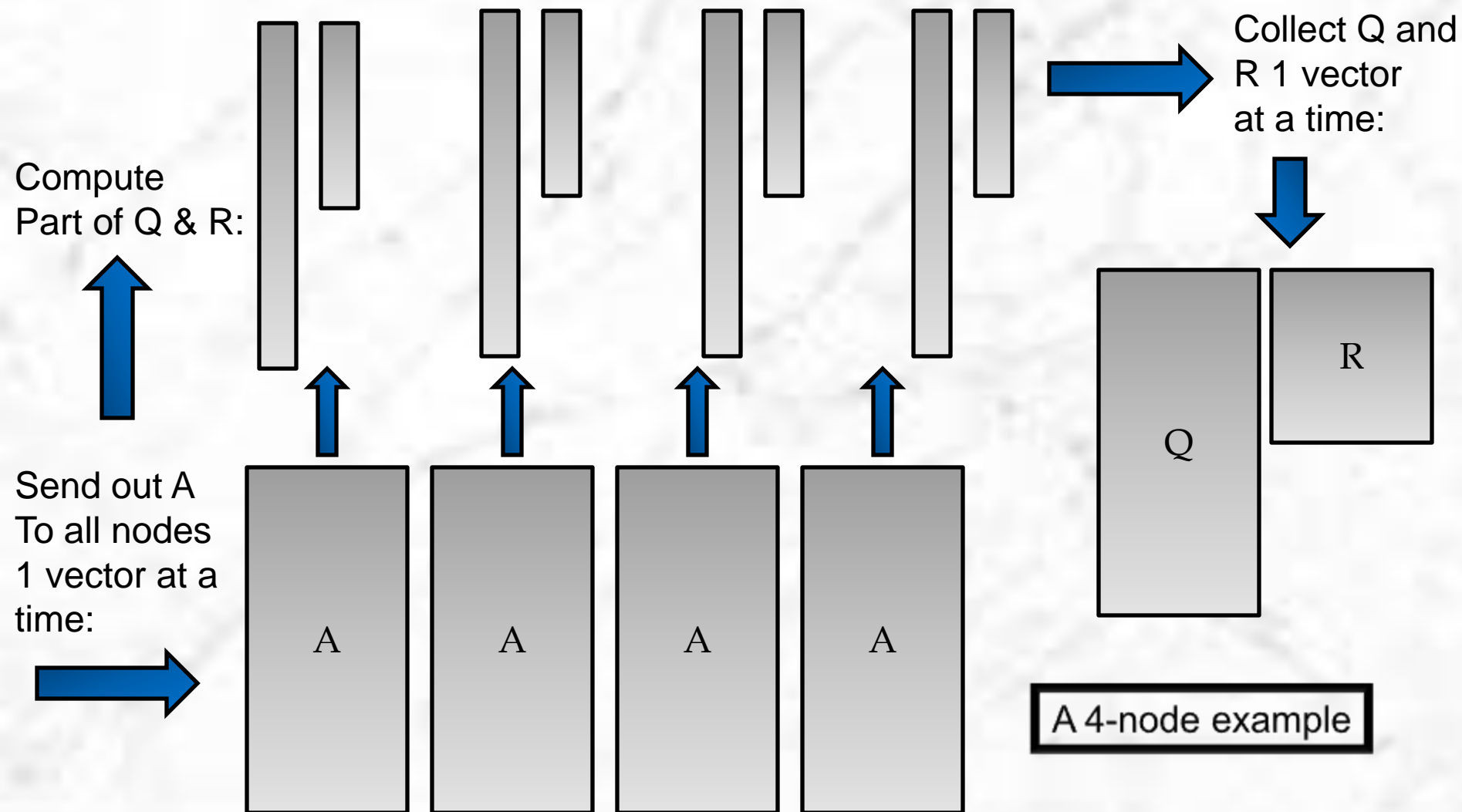


MATLAB vs. FORTRAN

Serial CGM QR-Factorization run times for $A(2N,N)$



Parallelization with MPI



Four Dual Quad-core Xeons with 4GB RAM per Node configured as 4 Processors Per Node for MPI/Batch (16 total processors)

Parallelization with MPI

```
call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, numnodes, ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)
```

Initialization.....

```
do i=1,N
    call MPI_BCAST(A(:,i), M, MPI_DOUBLE_PRECISION,
&                 0, MPI_COMM_WORLD, ierr)
enddo
```

Compute Q and R pieces...

```
do i=1,NpN
call MPI_GATHERV(Q(:,i), M, MPI_DOUBLE_PRECISION, Q(1,i),
&               rcountsM, displsM, MPI_DOUBLE_PRECISION, 0,
&               MPI_COMM_WORLD, ierr)
call MPI_GATHERV(R(:,i), N, MPI_DOUBLE_PRECISION, R(1,i),
&               rcountsN, displsN, MPI_DOUBLE_PRECISION, 0,
&               MPI_COMM_WORLD, ierr)
enddo
```

Validation....

```
call MPI_Finalize(ierr)
```

Parallel Computation Results

Calculate CGS QR factorization

.....PARALLEL.....

with 2 processors

Initializing A, Q, and R.....

...Done! Starting QR.....

Sending A matrix to all nodes.....

Nodes now computing 800 vectors of QR.

Computations completed, beginning to gather...

...Done! Validating.....

Relative Error in A: 9.398932753381588E-016

M: 3200 N: 1600

Number of Nodes: 2

Initialization time: 0.7603890 sec.

Communication time: 0.1733868 sec.

Computation time: 18.11910 sec.

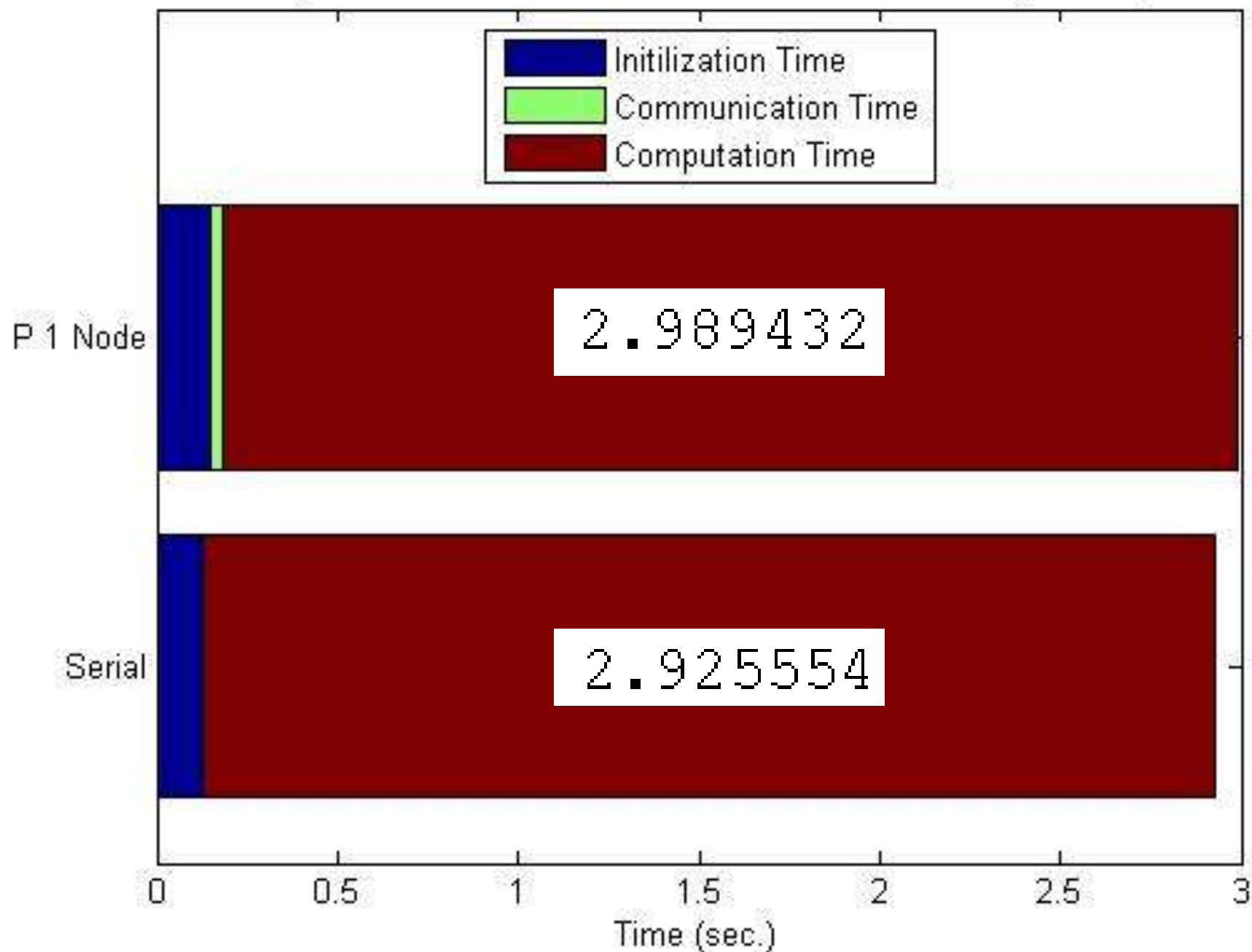
Validation time: 21.77310 sec.

Total time (w/o valid): 19.05287 sec.

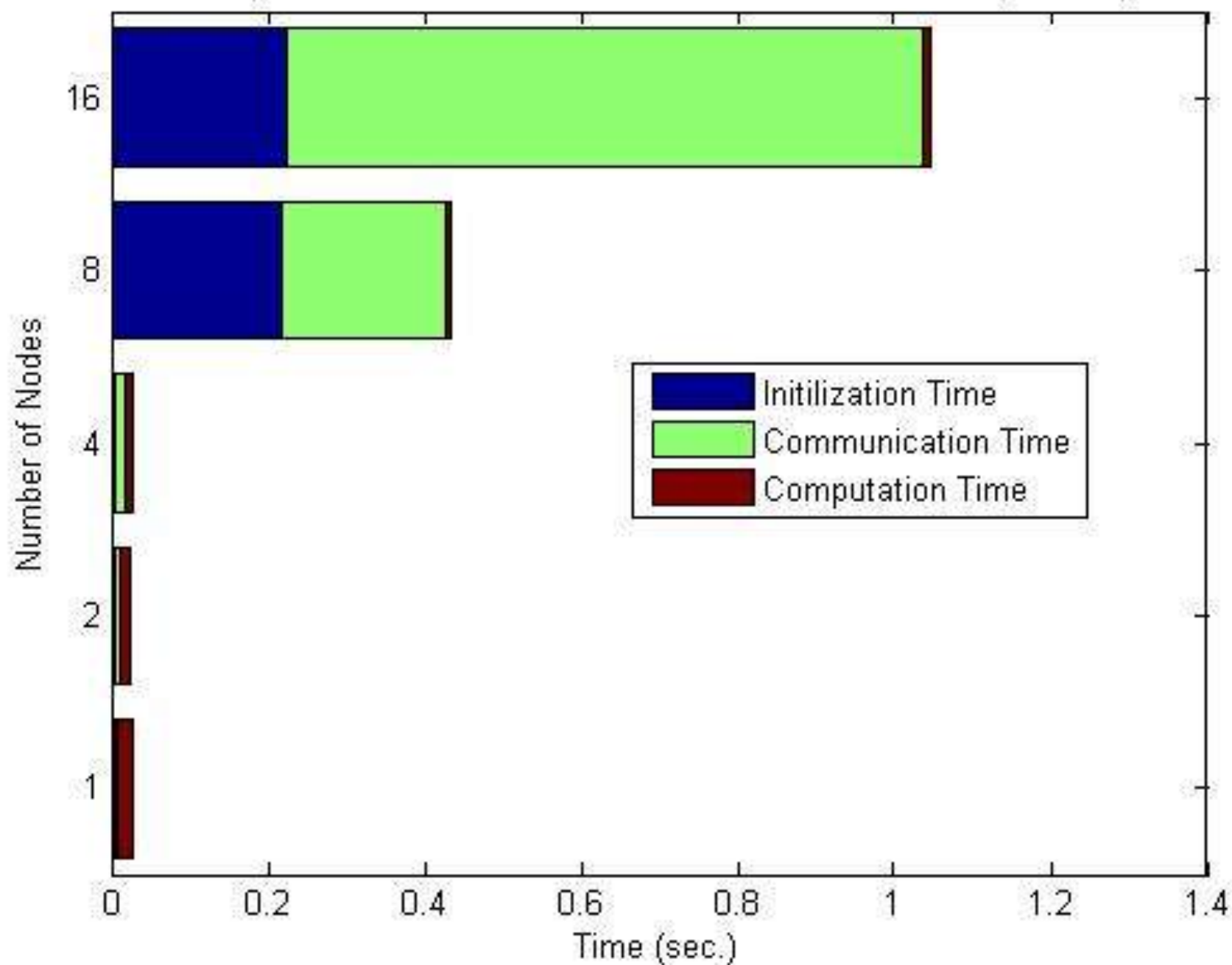
Using
Frobenius
Norm



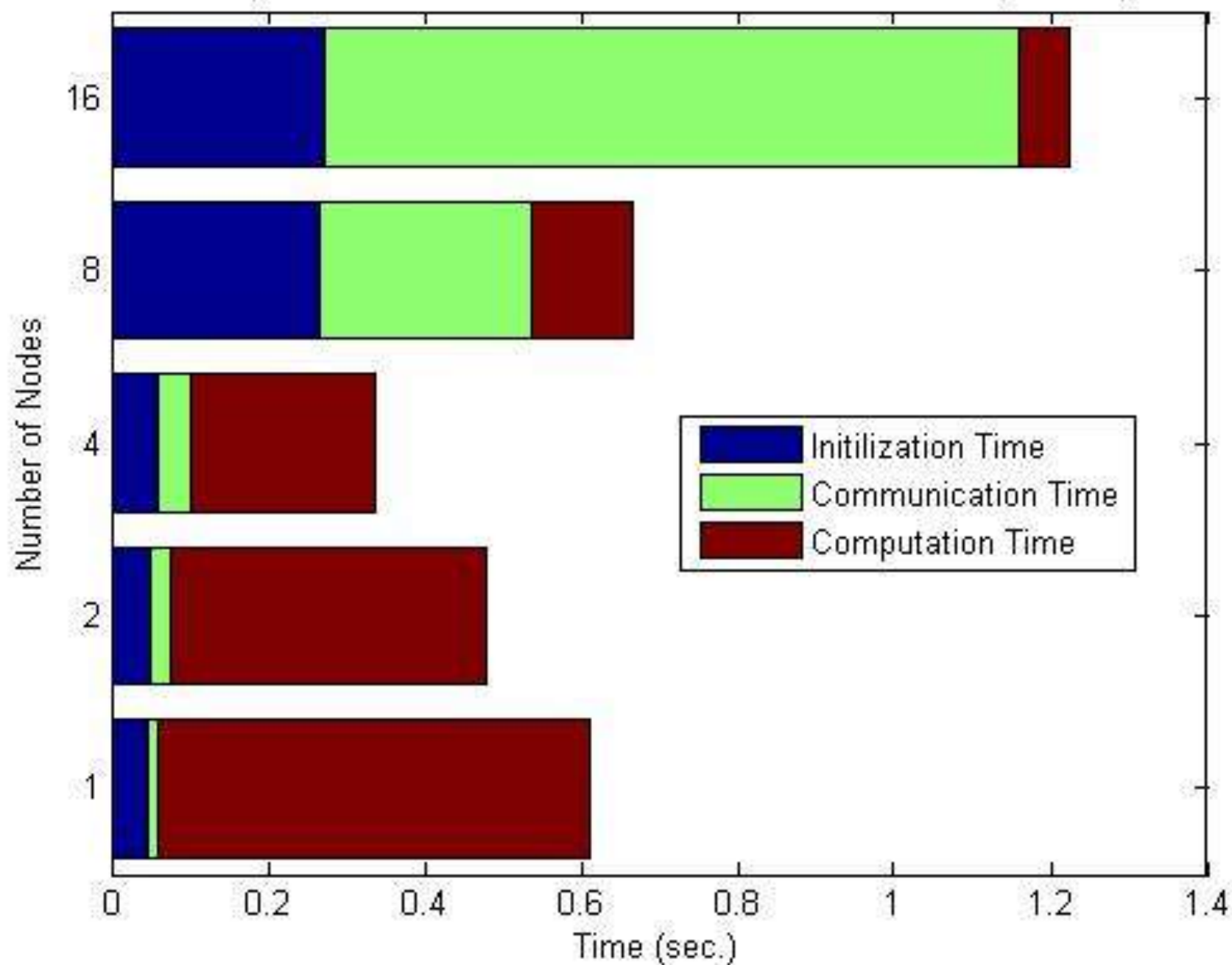
Computation time for Serial CGS QR-Factorization: A (1600,800)



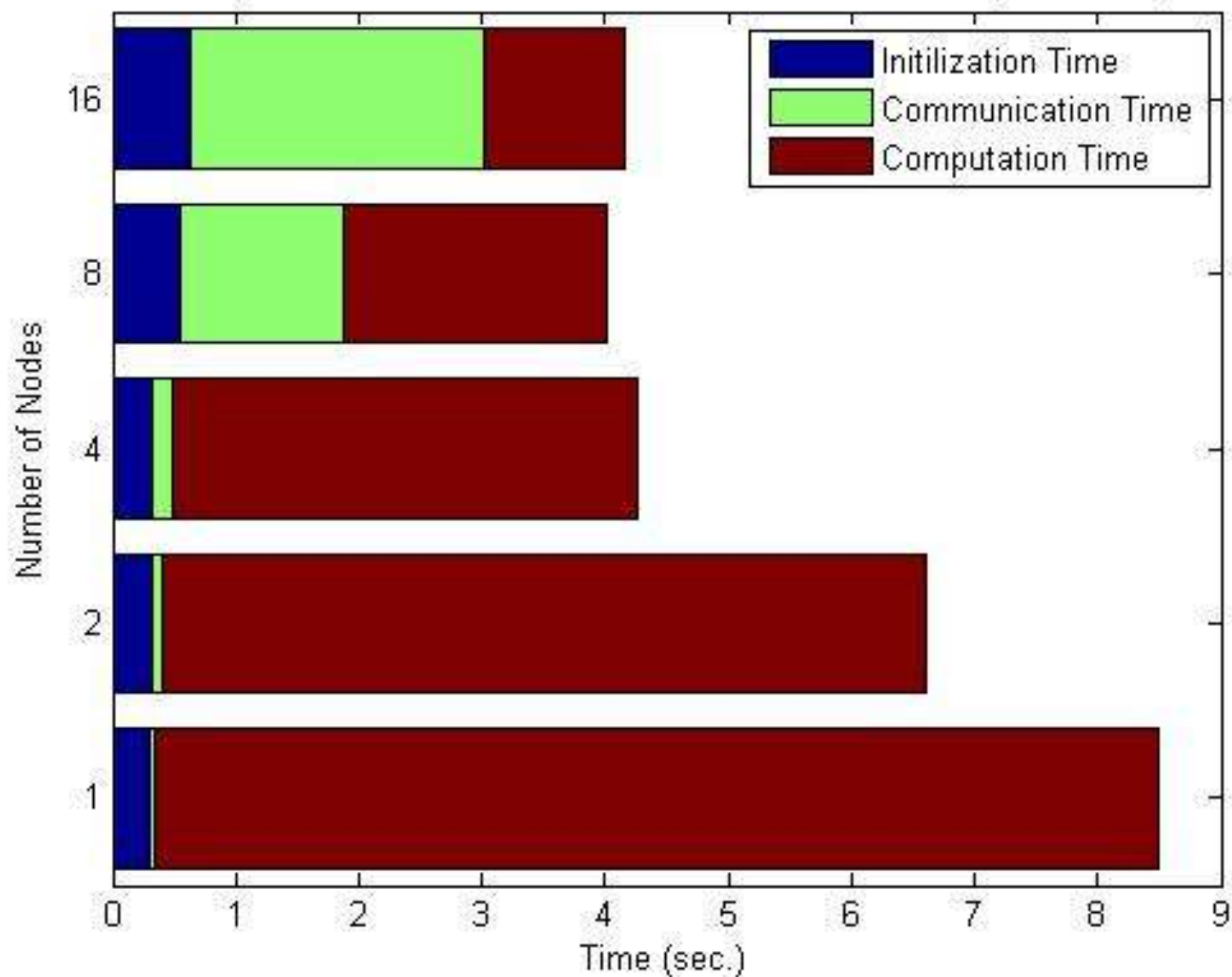
Computation time for Parallel CGS QR-Factorization: A (320,160)



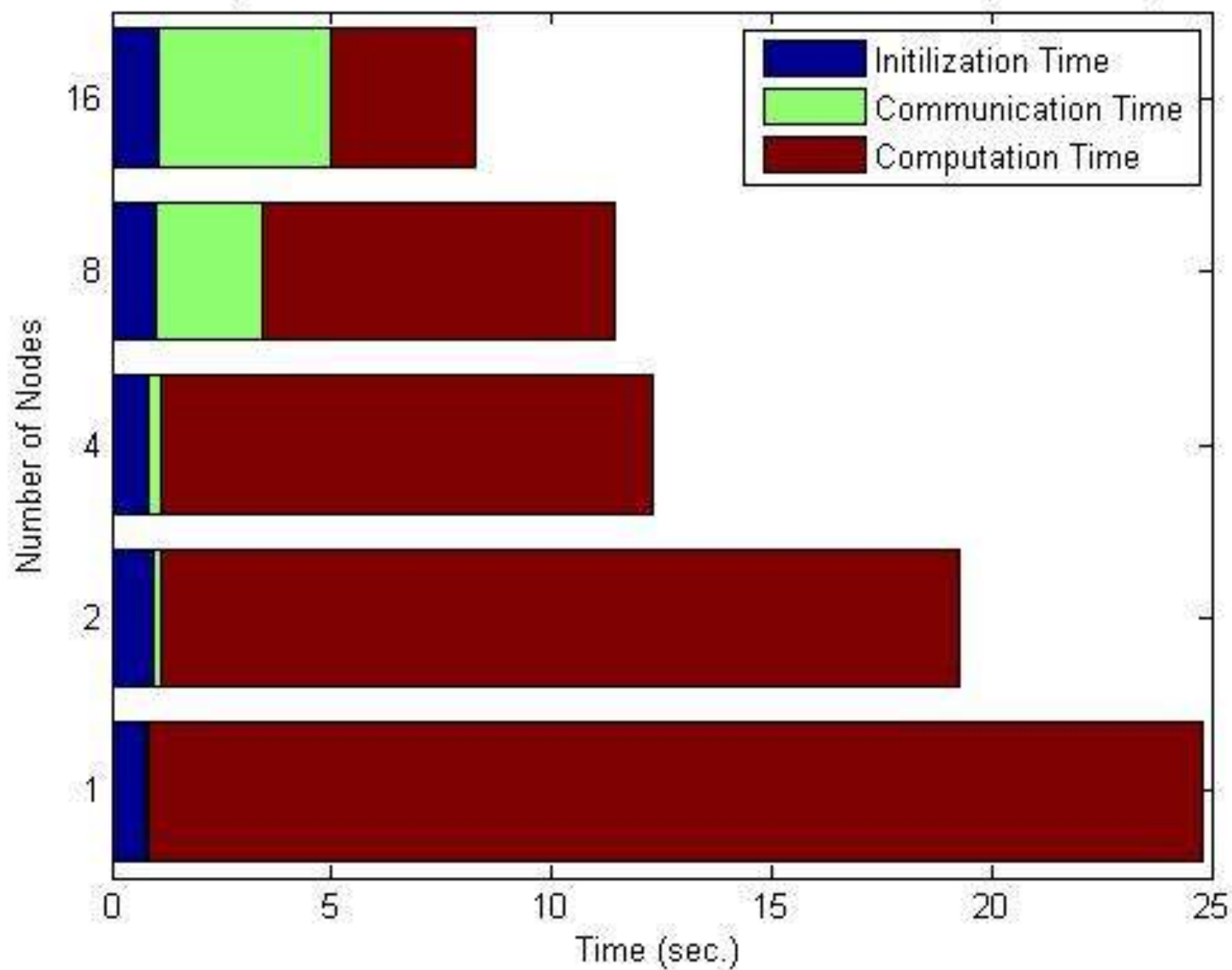
Computation time for Parallel CGS QR-Factorization: A (960,480)



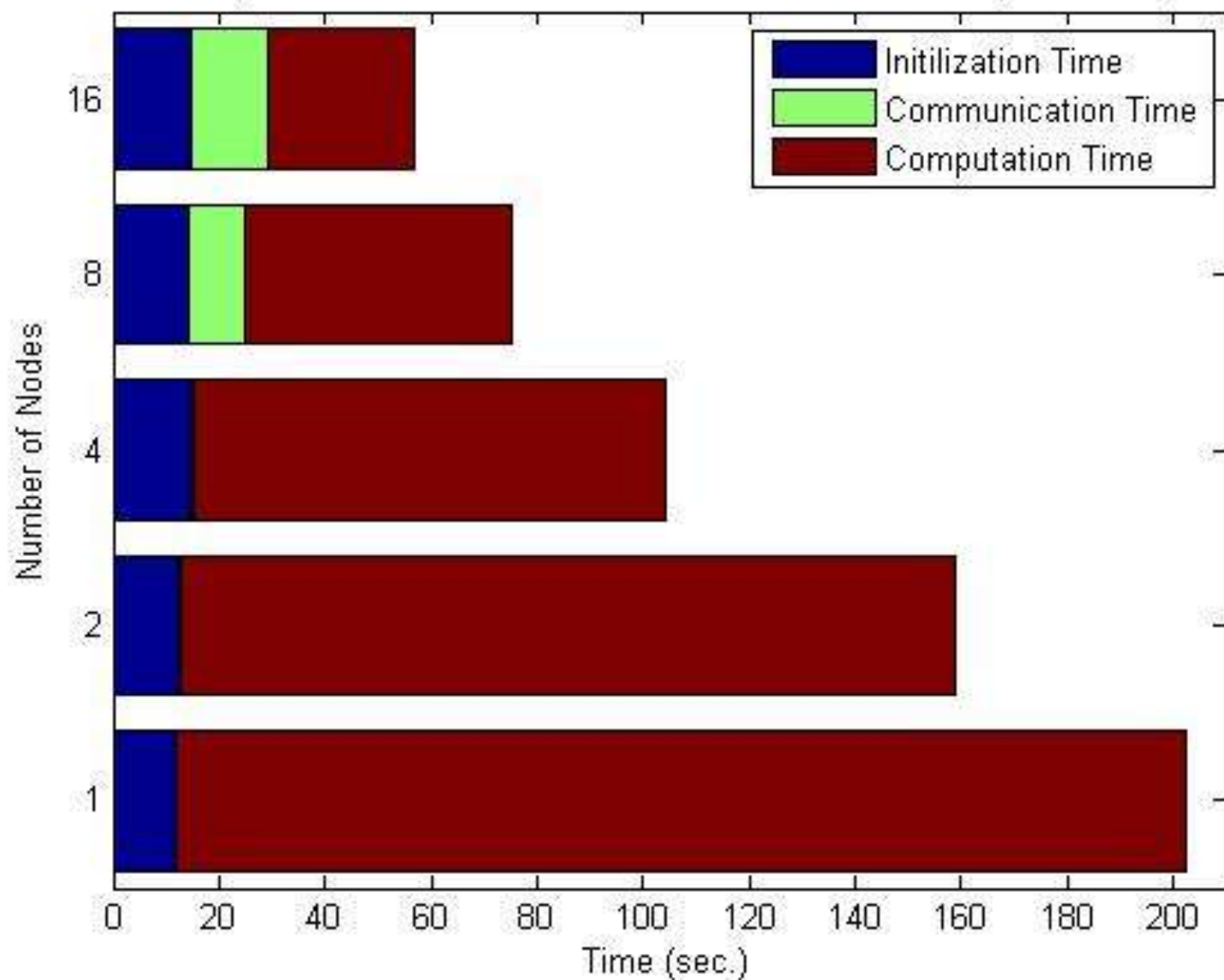
Computation time for Parallel CGS QR-Factorization: A (2240,1120)



Computation time for Parallel CGS QR-Factorization: A (3200,1600)



Computation time for Parallel CGS QR-Factorization: A (6400,3200)



Parallel Computation Results

Should dynamically change send packet size, and probably use sends and receives instead of broadcast and gather.... Why?

Because:

```
p0_15046: (3.781408) xx_shmalloc: returning NULL; requested 65576 bytes
p3_15049: (3.779917) xx_shmalloc: returning NULL; requested 65576 bytes
p1_15047: (3.779971) xx_shmalloc: returning NULL; requested 16424 bytes
p0_15046: (3.781497) p4_shmalloc returning NULL; request = 65576 bytes
```

```
P4_GLOBMEMSIZE (in bytes); the current size is 4194304
```

```
p3_15049: (3.780033) p4_shmalloc returning NULL; request = 65576 bytes
```

```
You can increase the amount of memory by setting the environment variable
```

```
p0_15046: p4_error: alloc_p4_msg failed: 0
```

```
p3_15049: p4_error: alloc_p4_msg failed: 0
```

```
p1_15047: p4_error: alloc_p4_msg failed: 0
```

Also, since CGS inner loop is from $i=1:j-1$, the distribution of work to Each node is not equal. This should be fixed as well.

Conclusion

- Parallelization of CGS does give nice speed-up for large matrices.
- Adding some improvements to the code such as non-uniform block sizes, and optimizing communication method would lead to even better speed-up.