Parallel Programming

OpenMP

Dr. Hyrum D. Carroll

November 22, 2016

Parallel Programming in a Nutshell

Load balancing vs Communication
This is the eternal problem in parallel computing. The basic approaches to this problem include:

- Data partitioning - moving different parts of the data set across several nodes
- Task partitioning - give separate tasks to different nodes

Definition of Terms

- node - a box usually containing processors, local memory, disks and network connection
- cluster - a group of nodes networked together
- speedup: $S_p = \frac{T_1}{T_p}$
- efficiency: $\frac{S_p}{p} = \frac{T_1}{pT_p}$

(T_i is the execution time for i processors, p is the number of processors)

Speedup

- Adding more processors does not always improve the speed a code runs.
- Usually, better speedup can be found by increasing the problem size, at least to a point.
- The non-parallel part of a code generally scales linearly with the problem size. The parallel part usually scales as problem size to some power.
- Generally increasing the problem size without increasing the node number helps performance.

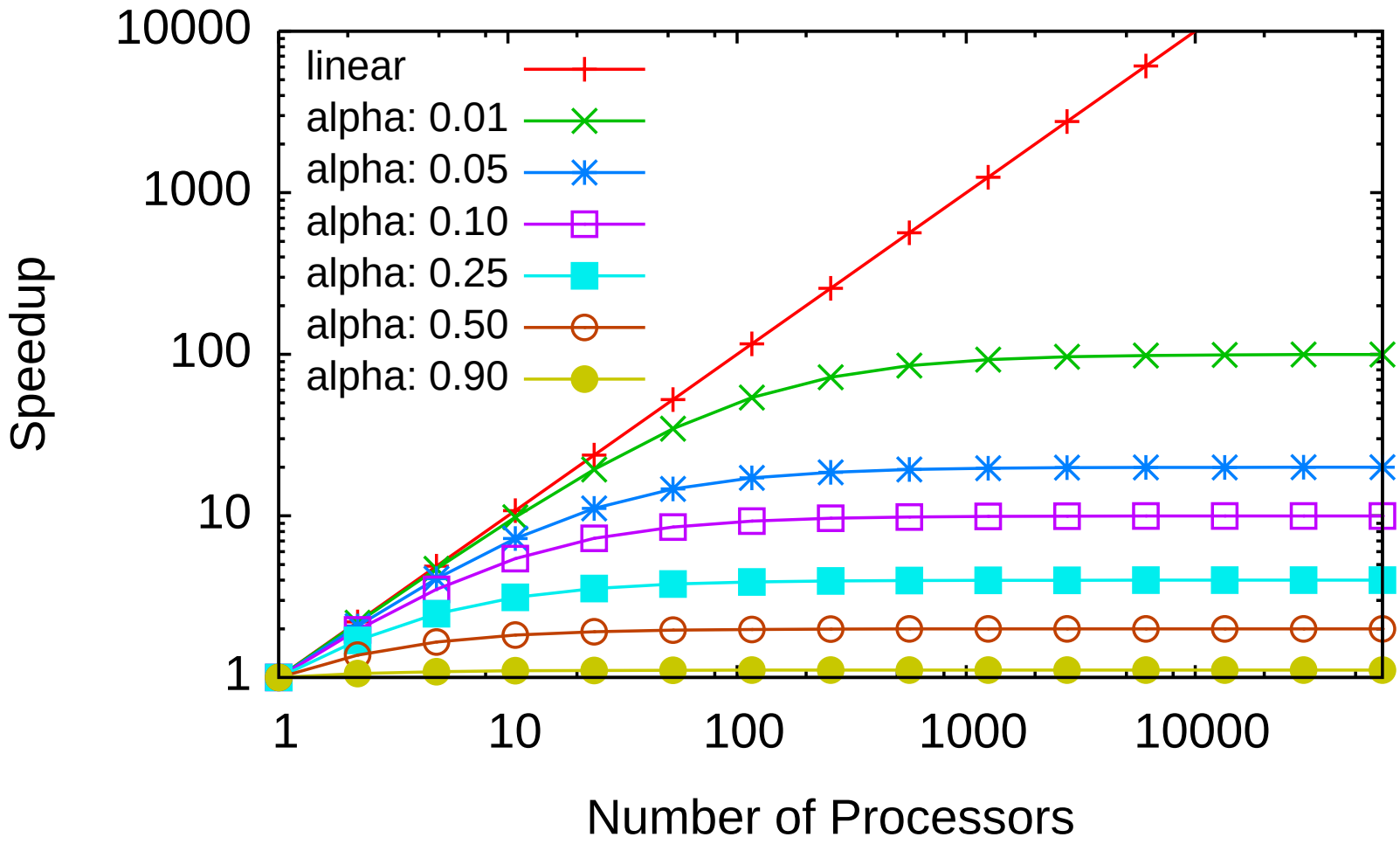
Scalability

Good parallel algorithms run faster when more nodes are available. In the best case, doubling the number of nodes decreases the execution time by a factor of two.
One way to consider scaling of a code is Amdahl's law:

$$\frac{1}{\alpha + \frac{1-\alpha}{p}}$$

where α is the portion of the code which cannot be parallelized and p is the number of processors. This is a simplification, but- **Speedup is limited by the slowest portion of the code.**

Amdahl's Law


10000
1000
100
10
1
Speedup
1
10
100
1000
10000
Number of Processors
linear
alpha: 0.01
alpha: 0.05
alpha: 0.10
alpha: 0.25
alpha: 0.50
alpha: 0.90

Communication

- Communication between nodes takes a great deal of time.
- Typically you can do thousands of computation in the time it takes to pass the simplest message.
- The time it takes for a message to be passed is limited by bandwidth b and latency l. To pass a message of size s, you need

$$\frac{s}{b} + l$$

(Assuming b, l, and s are in consistent units.)

Introduction to OpenMP

OpenMP is as a set of simple program additions to make codes run efficiently on shared memory computers. The formal API for OpenMP is only about 50 pages long, and contains compiler directives and library functions.

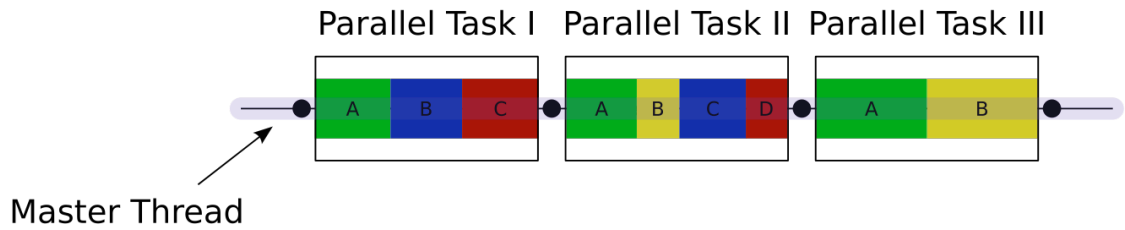
`http://www.llnl.gov/computing/tutorials/openMP/`

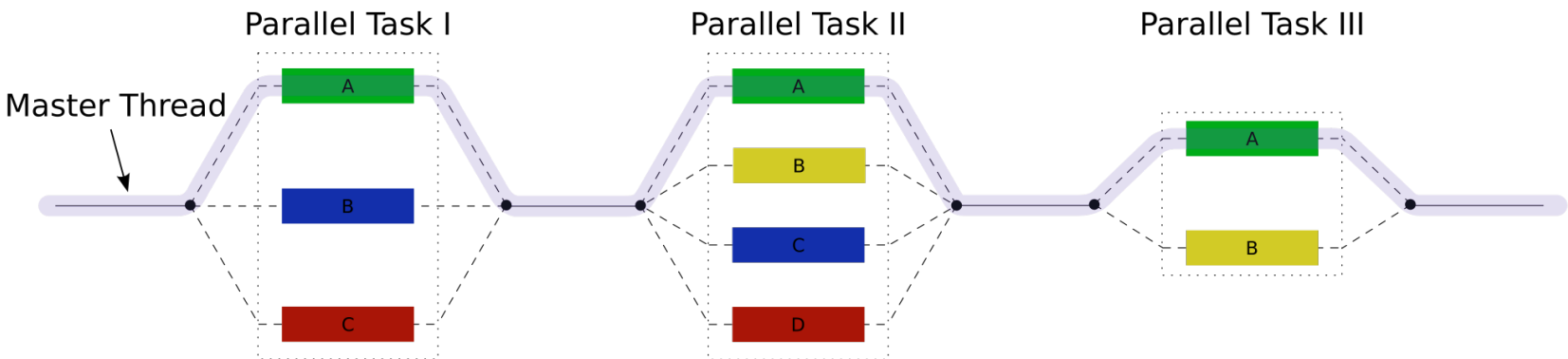
OpenMP Threads

OpenMP uses threads for parallel programming

- Forks and joins are used for most of the internal programming
- Speedup is achieved by the operating system splitting the threads across multiple CPUs.
- New threads are created explicitly by the program directives dynamically.

Forks and Joins





Goals of OpenMP - from LLNL

- Standardization
- Lean and Mean - only 3-4 directives
- Ease of use
- Portability - F77, F90, F95, C, C++

OpenMP Programming Model - from LLNL

- Shared Memory, thread based
- Explicit Parallelism
- Fork-Join Model
- Compiler Directives
- Nested Parallelism Support - in most implementations
- Dynamic Threads
- Not tied to I/O

Explicit Parallelism

- You must tell the computer what sections of code to parallelize using complier directives.
- The compiler directives vary between languages, but are ignored when OpenMP flags are not set with the compiler.
- Codes written with OpenMP can run easily on serial machines.

Environment and Library Routines

- Some environmental variables are needed to make the code execute using the correct number of threads
- Some library routines allow the programmer to set and access system variables

Not Message Passing

This is NOT a set of message passing routines. Instead, you give directives to the compiler of what parts of the code can be executed in parallel.
In some ways, OpenMP is a set of directives to tell the compiler how to more efficiently handle loops.

General Syntax

Fortran:

```
!$OMP <directive>
do useful stuff
!$OMP end <directive>
```

C/C++:

```
#pragma omp <directive-name> clause
{
do useful stuff in a structured block
}
```

A Trivial Example

Basic Code

```
1 program trivial
2   print*,'Hello World!'
3 end program
```

```
% gfortran trivial.f90
% ./a.out
 Hello World!
```

OMP Additions

```
1 program trivial
2
3 !$OMP PARALLEL
4   print*,'Hello World!'
5 !$OMP END PARALLEL
6
7 end program trivial
```

```
% gfortran trivialOpenMP.f90
% ./a.out
 Hello World!
```

A Trivial Example

Basic Code

```
1 program trivial
2   print*,'Hello World!'
3 end program
```

```
% gfortran trivial.f90
% ./a.out
 Hello World!
```

OMP Additions

```
1 program trivial
2
3 !$OMP PARALLEL
4   print*,'Hello World!'
5 !$OMP END PARALLEL
6
7 end program trivial
```

```
% gfortran trivialOpenMP.f90
% ./a.out
 Hello World!
```

What went wrong?

Execution of the Trivial Example

```
% gfortran trivialOpenMP.f90 -fopenmp
% ./a.out
 Hello World!
 Hello World!
 Hello World!
 Hello World!
 Hello World!
 Hello World!
 Hello World!
 Hello World!
% export OMP_NUM_THREADS=3
% ./a.out
 Hello World!
 Hello World!
 Hello World!
```

Thread ID

```
1  program trivial1
2    implicit none
3    integer :: OMP_GET_THREAD_NUM, OMP_GET_MAX_THREADS
4    integer :: tid, nthreads
5  !$OMP PARALLEL PRIVATE(nthreads, tid)
6    tid = OMP_GET_THREAD_NUM()
7    nthreads = OMP_GET_MAX_THREADS()
8    print*,'Hello World! from ', tid,nthreads
9  !$OMP END PARALLEL
10 end program
```

Note the PRIVATE key word, indicating that all threads have their own copy of the variable.

Thread ID (2)

```
% gfortran -fopenmp trivial1.f90
% ./a.out
 Hello World! from                0                1
 Hello World! from                2                1
 Hello World! from                3                1
 Hello World! from                4                1
 Hello World! from                1                1
 Hello World! from                7                1
 Hello World! from                5                1
 Hello World! from                6                1
```

Thread ID

```
1  program trivial2
2    implicit none
3    integer :: OMP_GET_THREAD_NUM, OMP_GET_MAX_THREADS
4    integer :: tid, nthreads
5    nthreads = OMP_GET_MAX_THREADS()
6  !$OMP PARALLEL PRIVATE(tid)
7    tid = OMP_GET_THREAD_NUM()
8    print*,'Hello World! from ', tid,nthreads
9  !$OMP END PARALLEL
10 end program
```

Note that `nthreads` is outside of the OMP directives

Thread ID (2)

```
% gfortran -fopenmp trivial2.f90
% ./a.out
 Hello World! from           5           8
 Hello World! from           0           8
 Hello World! from           1           8
 Hello World! from           2           8
 Hello World! from           7           8
 Hello World! from           3           8
 Hello World! from           4           8
 Hello World! from           6           8
```

Parallelizing Loops

To parallelize a loop, you need to help the compiler figure out the most efficient way to use threads. There are simple defaults, but giving it more details can help efficiency.
The basic directives are:

```
!$OMP PARALLEL
!$OMP DO

some parallel loop

!$OMP END DO
!$OMP END PARALLEL
```

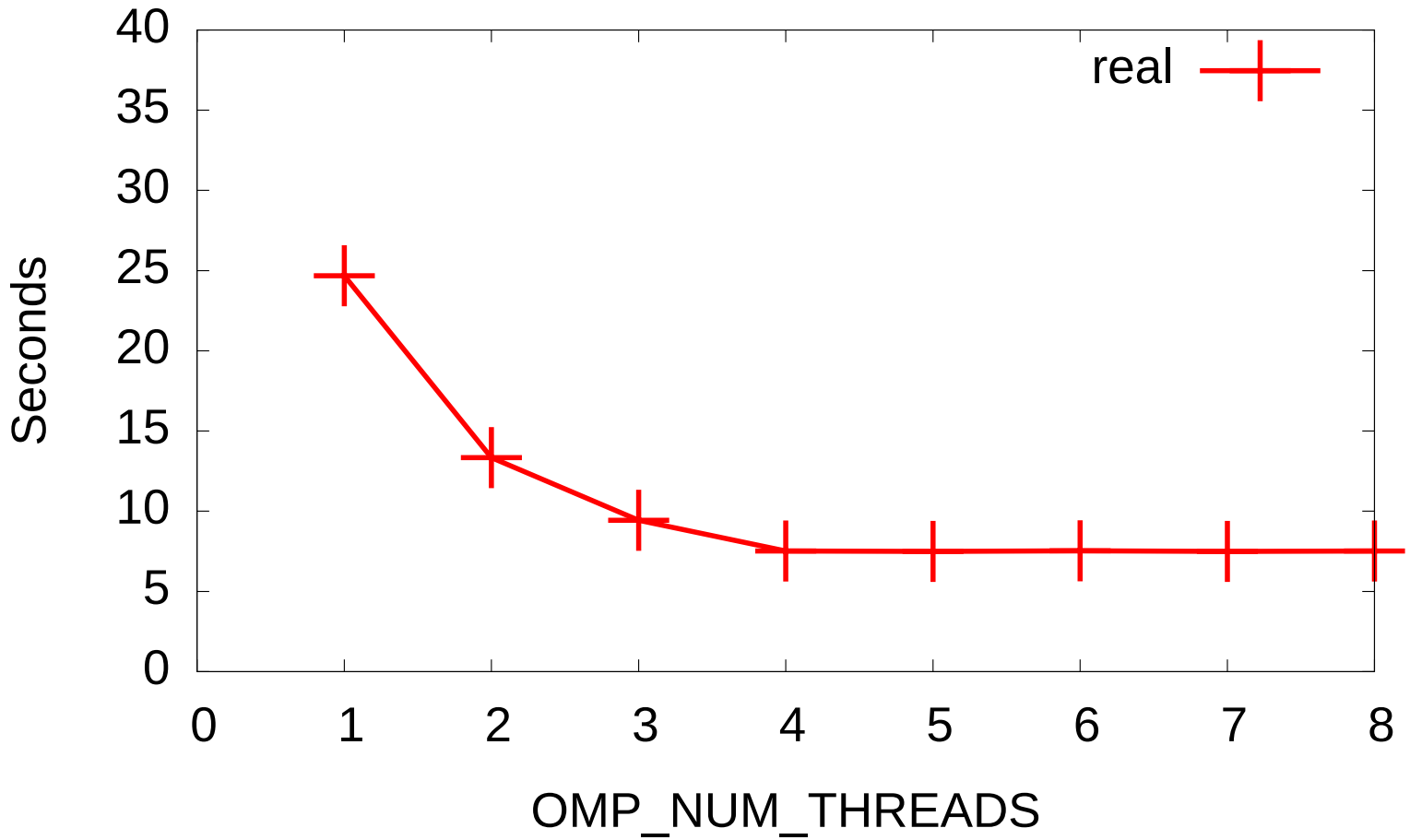
A Simple OMP Example

omptest1

```
1  program omptest1
2    integer, parameter :: n = 10000
3    integer, parameter :: dble = selected_real_kind
         (15,307)
4    real (kind=dble), dimension(n) :: a
5    integer :: i, j
6  !$OMP PARALLEL
7  !$OMP DO
8    do j = 1, 100000
9       do i = 1, n
10         a(i) = log(real(i)) + j
11      enddo
12   enddo
13 !$OMP END DO
14 !$OMP END PARALLEL
15   print*,a(1)
16 end program omptest1
```

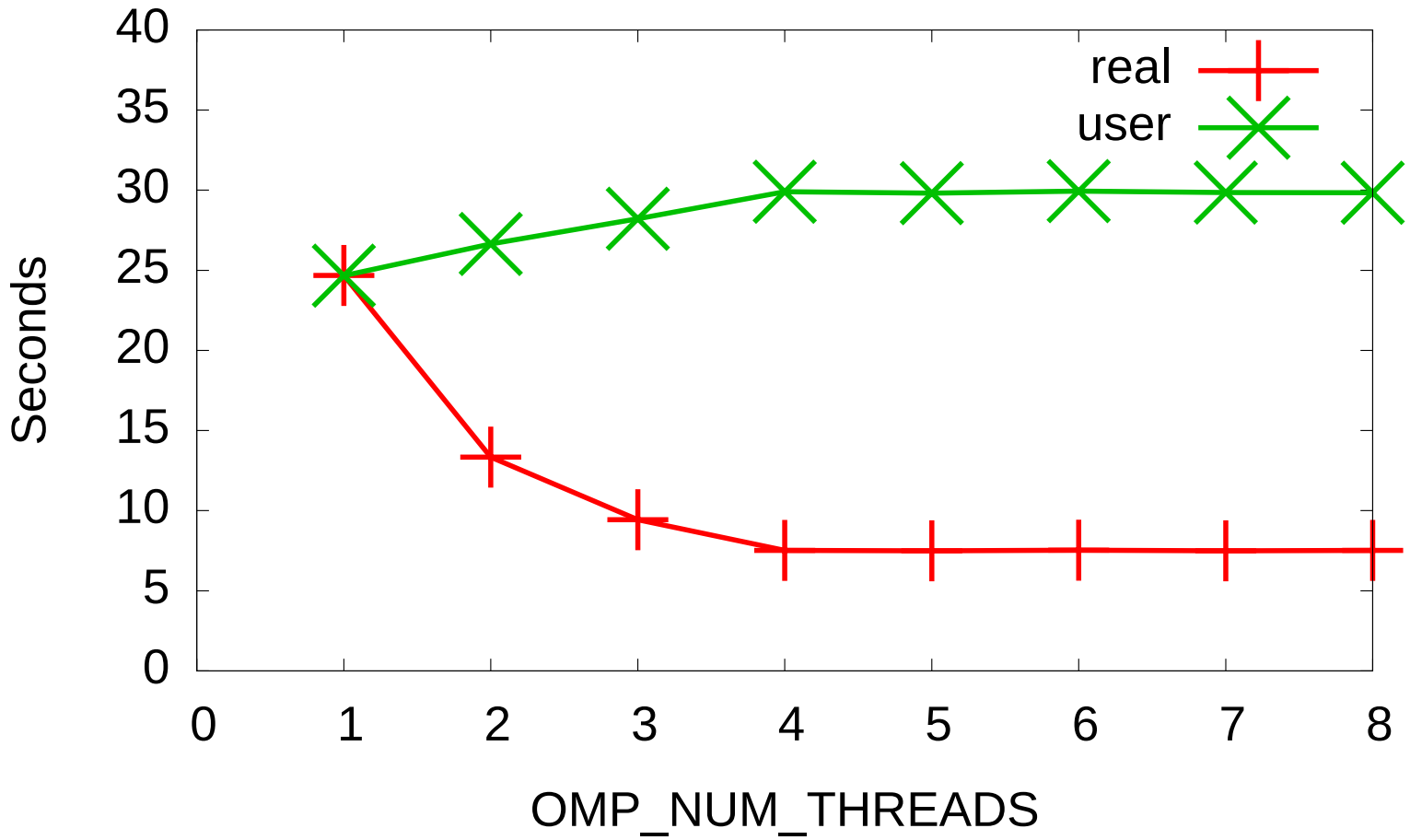
Results

omptest1



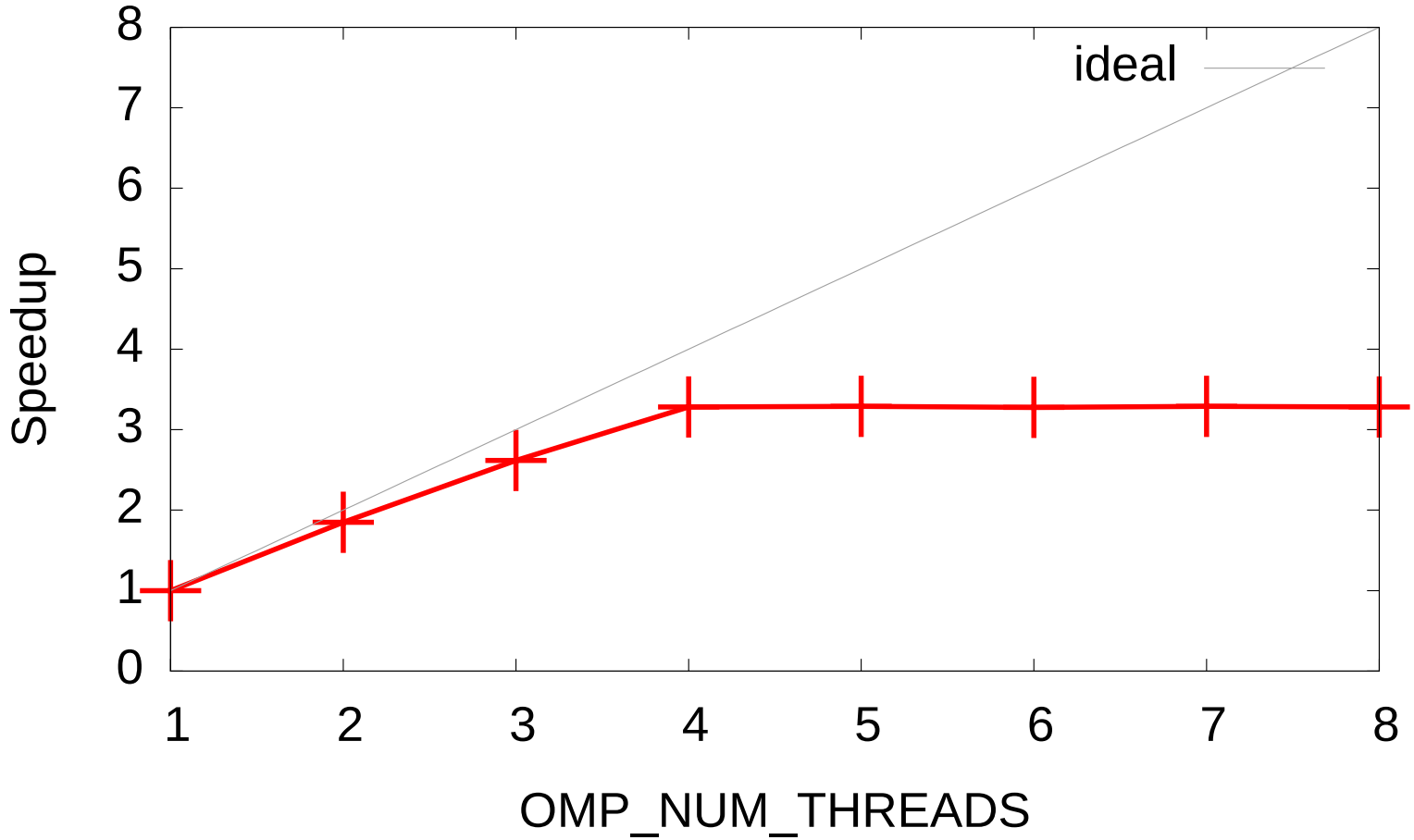
Results

omptest1



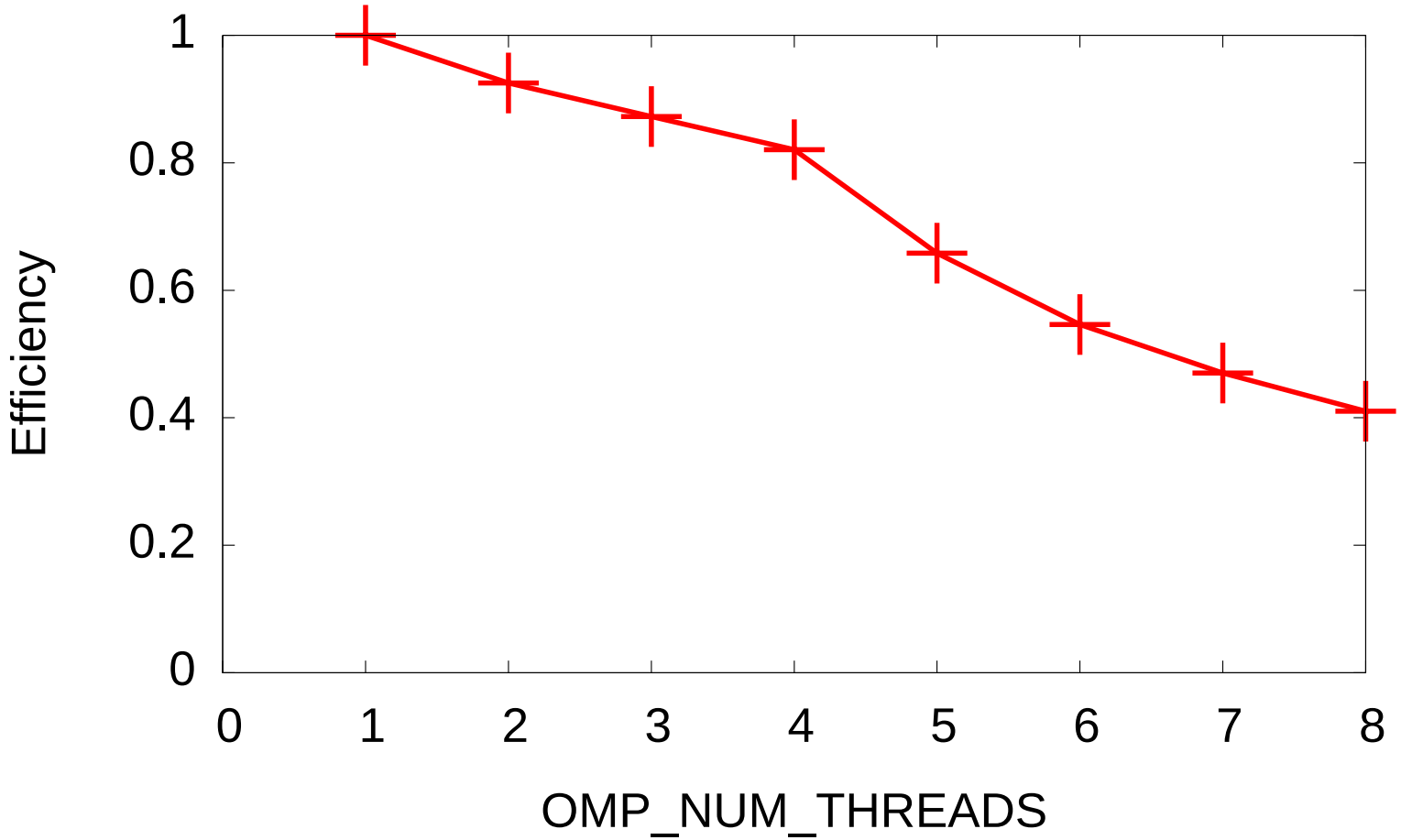
Results

omptest1



Results

omptest1



Combining Directives

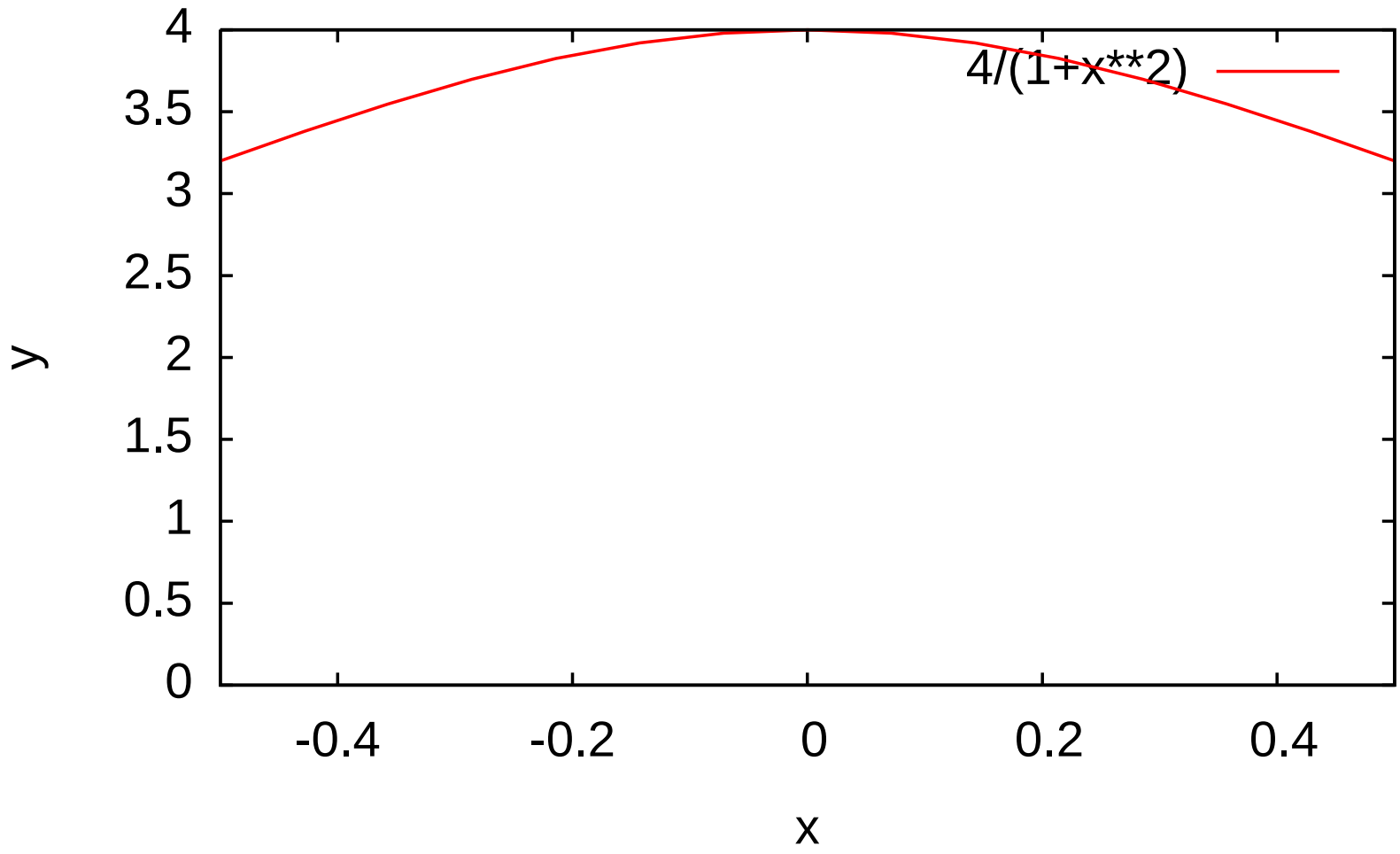
You do not have to have a separate directive on each line. For example,

```
!$OMP PARALLEL
!$OMP DO
!$OMP PRIVATE(NTHREADS, TID)
```

Becomes

```
!$OMP PARALLEL  DO  PRIVATE(NTHREADS, TID)
```

Numerical Integration



Numerical Integration

Integrating

$$\pi = \int_{-1/2}^{1/2} \frac{4}{1+x^2} dx \tag{1}$$

We can approximate this integral using Simpson's algorithms

- Input the number of partitions to be used
- Divide the domain into n partitions
- Evaluate the function at each partition
- Multiply the function evaluation times the width of the function to find a differential area
- Add the differential areas together
- Output the result

Parallel Integration

In parallel, the problem is nearly the same.

- Have processing element (PE) zero, get the number of partitions, n
- Determine the number of PEs: m
- Divide the domain into $\frac{n}{m}$ partitions on each PE
- Evaluate the function at each partition
- Multiply the function evaluation times the width of the function to find a differential area
- Add the differential areas together across all the PEs
- On PE zero, output the result

Simple Code to Calculate PI

```
1  program reduce
2    integer :: i, num_steps
3    double precision :: x, pi, step, sum
4    sum =0.0d0; nsteps = 10000
5    step = 1.0d0 / dble(nsteps)
6    do i = 1, nsteps
7       x = (dble(i) + 0.5d0) * step
8       sum = sum + 4.0d0 /(1.0d0 + x*x)
9    enddo
10   pi = step * sum
11   print *, "Estimate of Pi with ", nsteps, " steps is "
         , pi
12 end program reduce
```

```
$ ./reduce
 Estimate of Pi with  10000 steps is  3.1413926444243838
```

Simple Code to Calculate PI

```
1  program reduceOMP
2    integer :: i, num_steps
3    double precision :: x, pi, step, sum
4    sum =0.0d0; num_steps = 10000
5    step = 1.0d0 / dble(num_steps)
6  !$OMP PARALLEL DO
7    do i = 1, num_steps
8       x = (dble(i) + 0.5d0) * step
9       sum = sum + 4.0d0 /(1.0d0 + x*x)
10   enddo
11 !$OMP END PARALLEL DO
12   pi = step * sum
13   print *, "Estimate of Pi with ", num_steps, " steps
         is ", pi
14 end program reduceOMP
```

```
$ gfortran -fopenmp reduceOMP.f90 -o reduceOMP
$ ./reduceOMP
 Estimate of Pi with  10000 steps is  7.558833581770253
```

Simple Code to Calculate PI

```
1  program reduceOMP
2    integer :: i, num_steps
3    double precision :: x, pi, step, sum
4    sum =0.0d0; num_steps = 10000
5    step = 1.0d0 / dble(num_steps)
6  !$OMP PARALLEL DO
7    do i = 1, num_steps
8       x = (dble(i) + 0.5d0) * step
9       sum = sum + 4.0d0 /(1.0d0 + x*x)
10   enddo
11 !$OMP END PARALLEL DO
12   pi = step * sum
13   print *, "Estimate of Pi with ", num_steps, " steps
         is ", pi
14 end program reduceOMP
```

```
$ gfortran -fopenmp reduceOMP.f90 -o reduceOMP
$ ./reduceOMP
 Estimate of Pi with  10000 steps is  7.558833581770253
```

What happened?

Reductions

Because the loops are executing separately, you may wish to combine the results from different threads to a final answer. You need to use reduction to make this work.

```
$!OMP PARALLEL PRIVATE(X) REDUCTION(+:SUM)
```

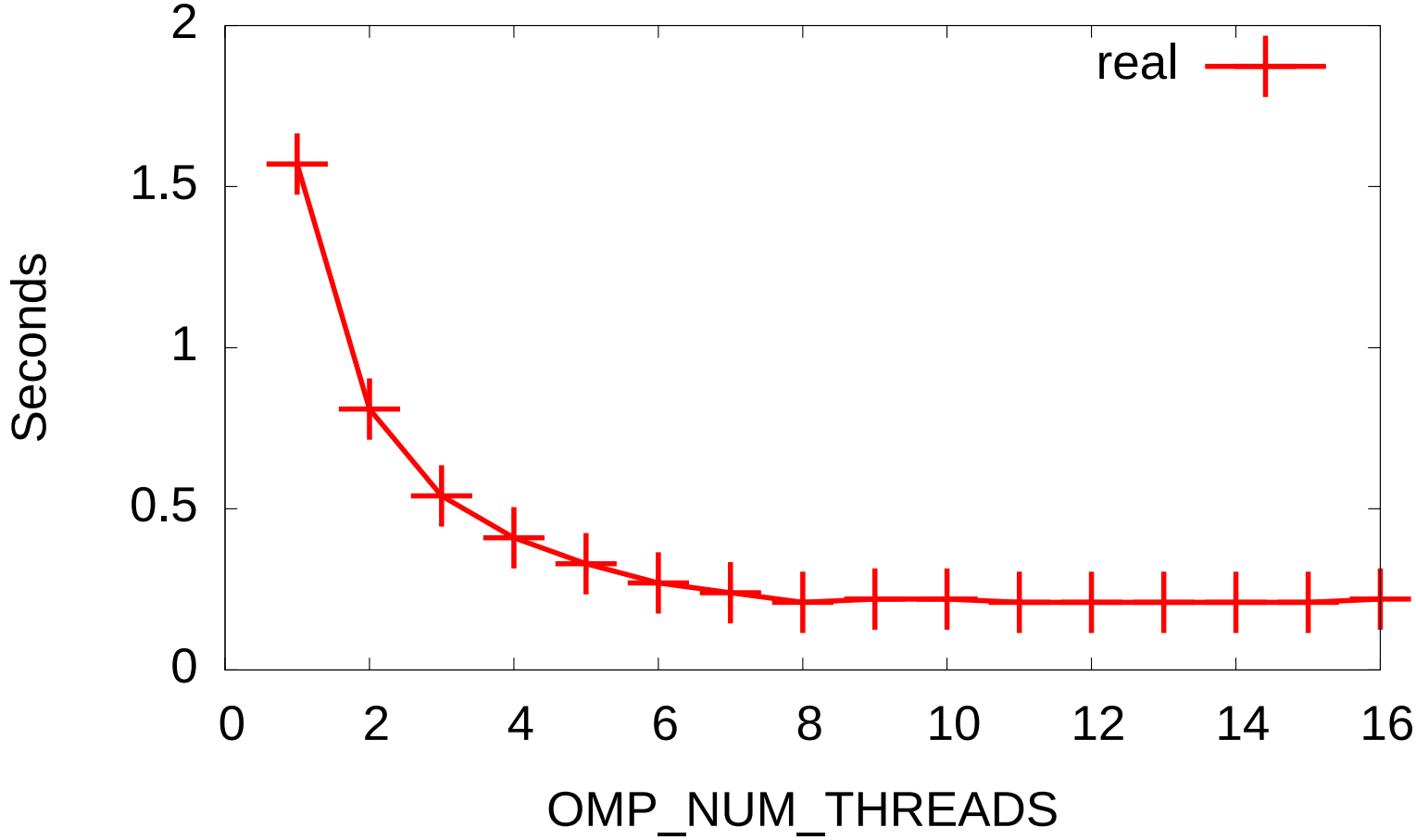
OpenMP Modifications

```
1  program reduceOMP2
2    integer :: i, num_steps
3    double precision :: x, pi, step, sum
4    sum =0.0d0 ; nsteps = 100000000
5    step = 1.0d0 / dble(nsteps)
6  !$OMP PARALLEL DO PRIVATE(X) REDUCTION(+:SUM)
7  do i = 1, nsteps
8    x = (dble(i) + 0.5d0) * step
9    sum = sum + 4.0d0 /(1.0d0 + x*x)
10 enddo
11 !$OMP END PARALLEL DO
12   pi = step * sum
13   print *, "Estimate of Pi with ", nsteps, " steps is "
         , pi
14 end program reduceOMP2
```

```
$ gfortran -fopenmp reduceOMP2.f90 -o reduceOMP2
$ ./reduceOMP2
 Estimate of Pi with  10000 steps is  3.1413926444243732
```

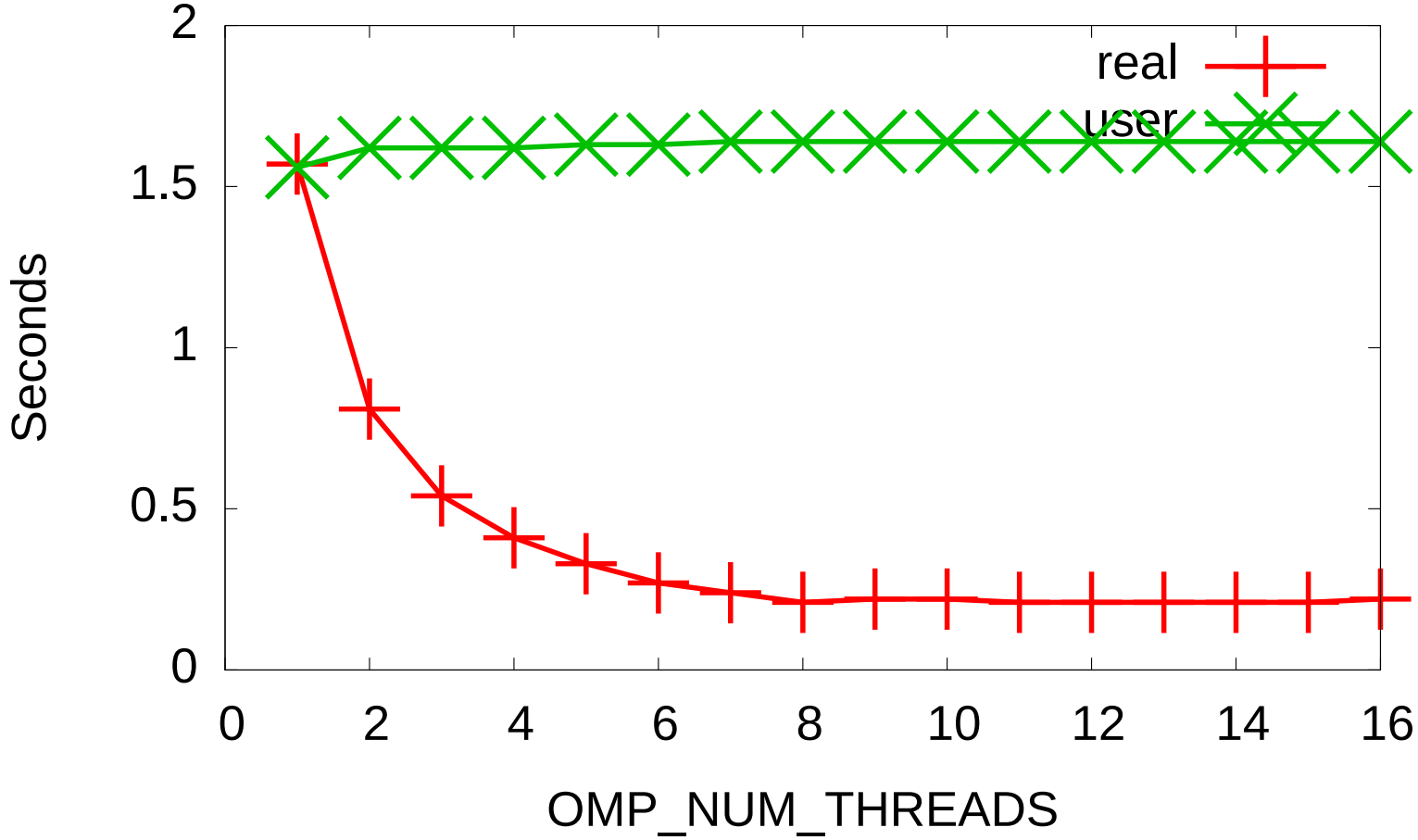
Results

reduceOMP2



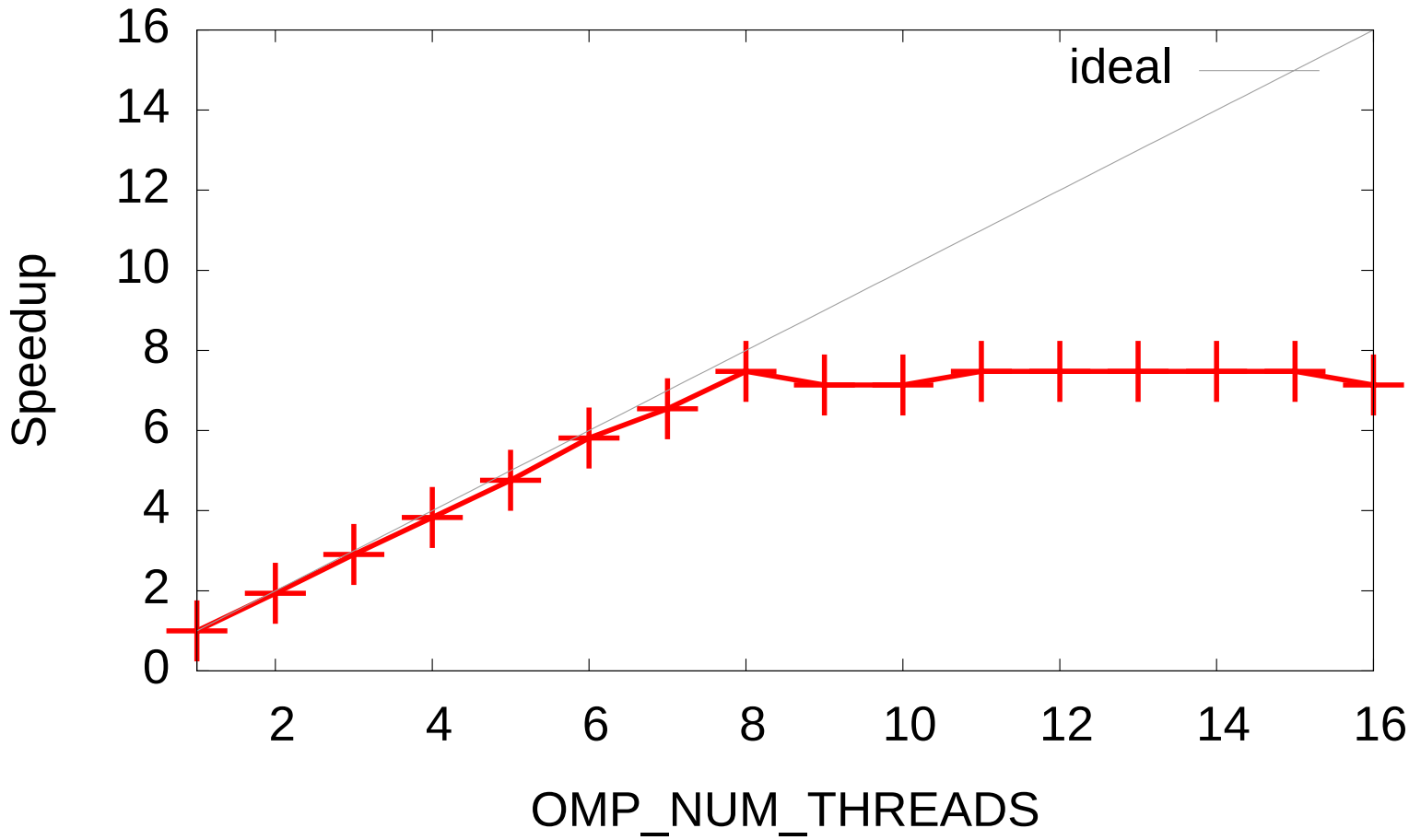
Results

reduceOMP2



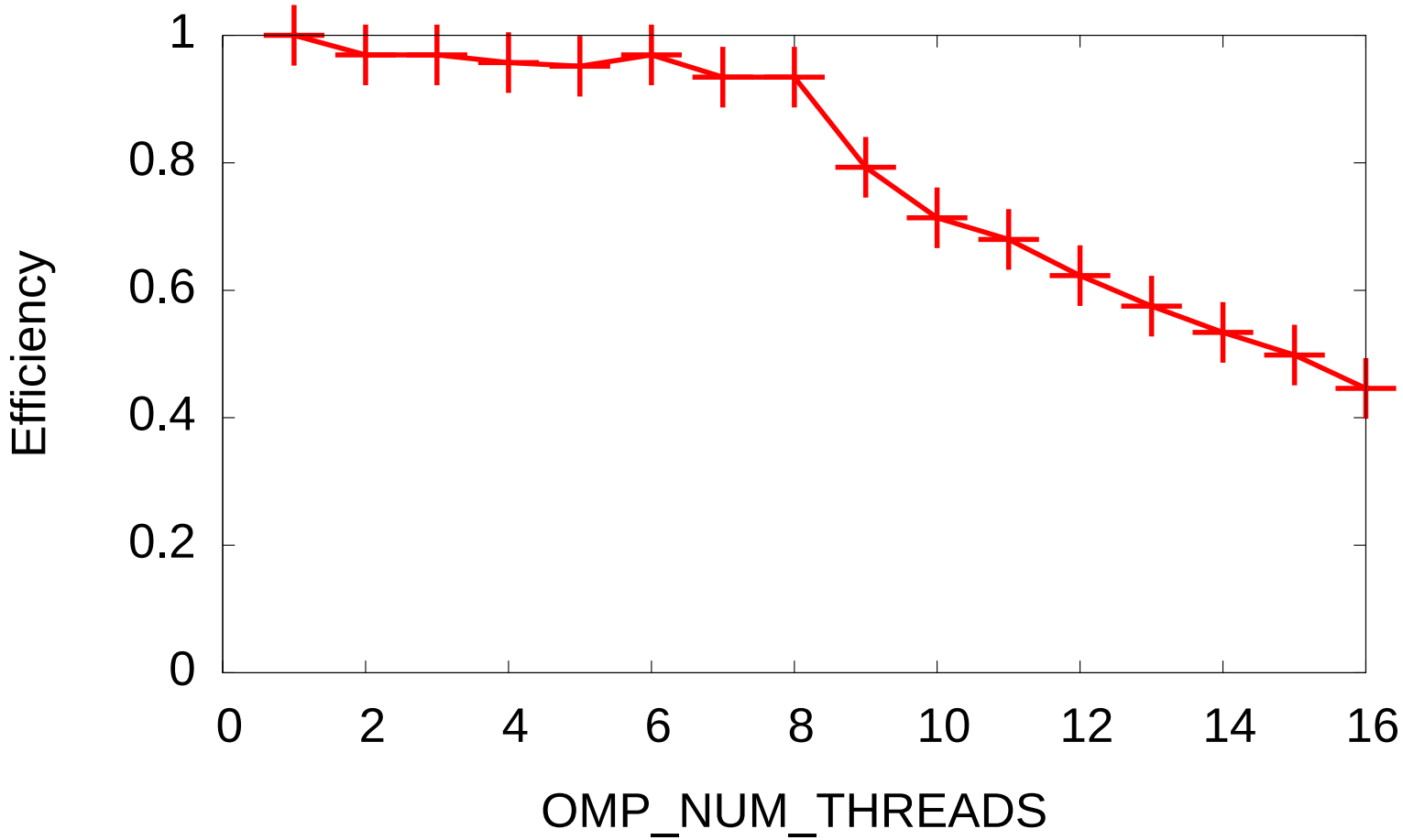
Results

reduceOMP2



Results

reduceOMP2



Loop Splitting

One of the key ideas to remember is that loops often contain several operations that can be split. Taking an example from the Patterns in Parallel Programming book, imagine we have a loop with two functions:

- *BIG_COMPUTATION* - a big computation the executes independently on each element in the loop
- *COMBINE* - an element that cannot be parallelized and must execute in order

Loop Splitting

```
do i = 1, nsteps
  x = BIG_COMPUTATION(i)
  call COMBINE(x,answer)
enddo
```

can be split into

```
do i = 1, nsteps
  x(i) = BIG_COMPUTATION(i)
enddo
do i = 1, nsteps
  call COMBINE(x(i),answer)
enddo
```

Using OpenMP in Loop Splitting

```
!$OMP PARALLEL DO PRIVATE(I)
do i = 1, nsteps
  x(i) = BIG_COMPUTATION(i)
enddo
!$OMP END PARALLEL DO

do i = 1, nsteps
  call COMBINE(x(i),answer)
enddo
```

Controlling Loops

There are many options for controlling the execution of threads.

```
!$OMP DO SCHEDULE(TYPE,integer)
```

- schedule(static[,chunk]) - groups of size `chunk` statically assigned in a round-robin fashion
- schedule(dynamic[,chunk]) - threads dynamically grab work as it is completed
- schedule(guided[,chunk]) - chunk size is reduced automatically during iteration toward a minimum level of `chunk`
- schedule(runtime) - checks the OMP_SCHEDULE environmental variable

Controlling Loops

```
integer, parameter :: chunk = 10

!$OMP PARALLEL PRIVATE(i,j,z,c,it) DEFAULT(SHARED)
!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
  do i = 1, n
    do j = 1, n
...
```

Controlling Loops

```
setenv OMP_SCHEDULE static
11.477u 0.012s 0:08.24 139.3%

setenv OMP_SCHEDULE dynamic
11.239u 0.006s 0:05.67 198.0%

setenv OMP_SCHEDULE guided
11.453u 0.005s 0:06.52 175.6%

setenv OMP_SCHEDULE static,20
11.439u 0.028s 0:05.89 194.3%

no omp
11.280u 0.004s 0:11.28 100.0%
```