OpenMP

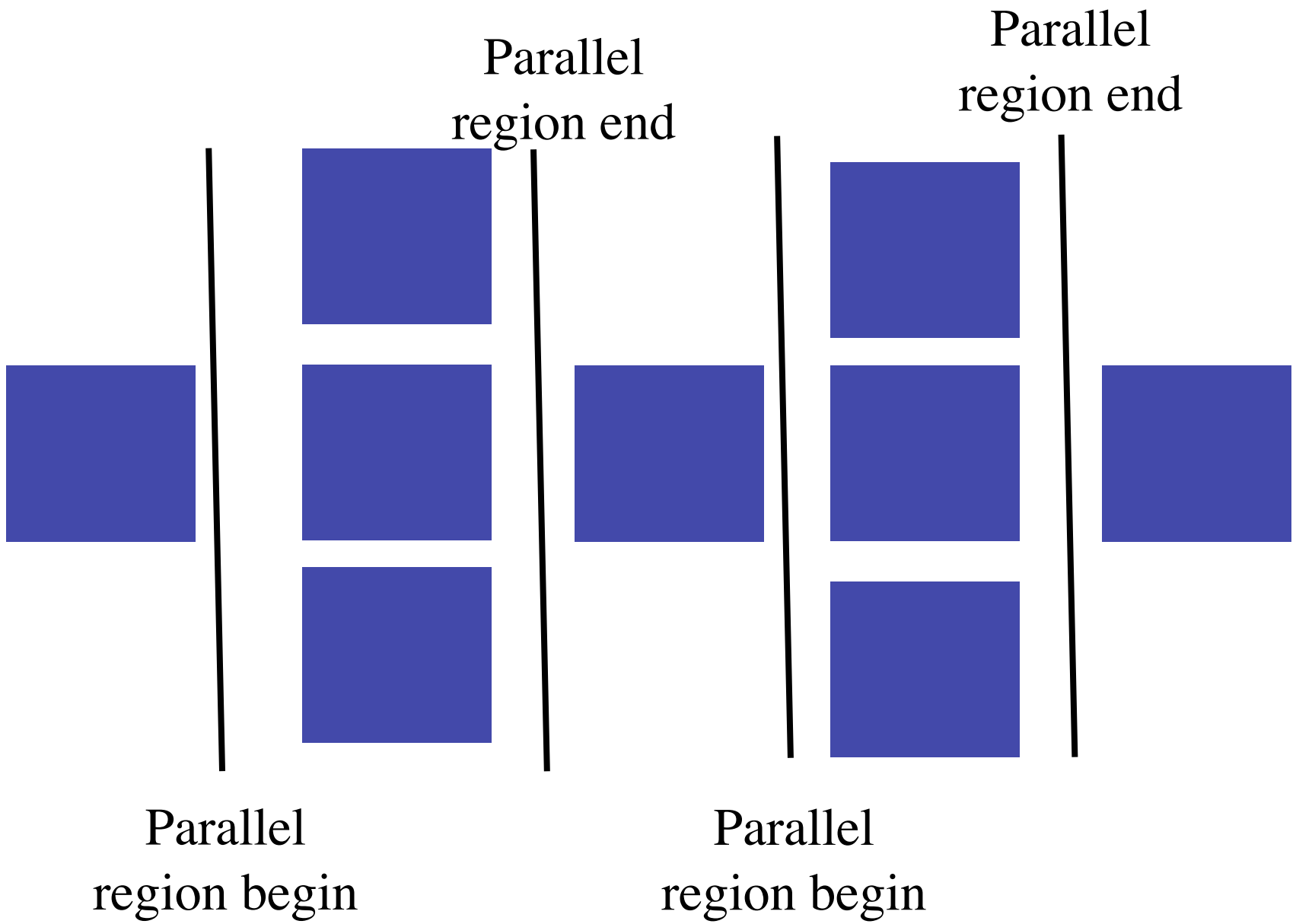
OpenMP: A little history

- Standard introduced in 1996
  - An industry standard after multiple companies and extended their own compilers to virtually the same tasks
  - Most of the predecessors offered more flexibility then the OpenMP standard
- Extension built into most commercial Fortran90, C, and C++ compilers

OpenMP

- The basics
- Loops
- Parallel environment
- Examples

OpenMP program diagram



Hello world

```
program hello_world
write(0,*) “Hello world”
end program
```

```
>ifort hello.f90
>./a.out
Hello world
```

```
#include<stdio.h>
int main (int argc, char **argv)
{
  fprintf(stderr,”Hello world”);
}
```

```
>icc hello.c
>./a.out
Hello world
```

Hello world

```
program hello_world
!$OMP PARALLEL
write(0,*) “Hello world”
!$OMP END PARALLEL
end program
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world
Hello world
Hello world
```

```
#include<stdio.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,”Hello world”);
}
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world
Hello world
Hello world
```

Hello world

```
program hello_world
!$OMP PARALLEL
write(0,*) “Hello world”
!$OMP END PARALLEL
end program
```

```
#include<stdio.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,”Hello world”);
}
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world
Hello world
Hello world
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world
Hello world
Hello world
```

Begin and end a parallel environment in Fortran90

Hello world

```
program hello_world
!$OMP PARALLEL
write(0,*) “Hello world”
!$OMP END PARALLEL
end program
```

```
#include<stdio.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,”Hello world”);
}
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world
Hello world
Hello world
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world
Hello world
Hello world
```

Delineate a parallel region in C

Hello world

```
program hello_world
!$OMP PARALLEL
write(0,*) “Hello world”
!$OMP END PARALLEL
end program
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world
Hello world
Hello world
```

```
#include<stdio.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,”Hello world”);
}
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world
Hello world
Hello world
```

Tell how many different copies of the programs to run simutaneously

Hello world

```
program hello_world
!$OMP PARALLEL
write(0,*) “Hello world”
!$OMP END PARALLEL
end program
```

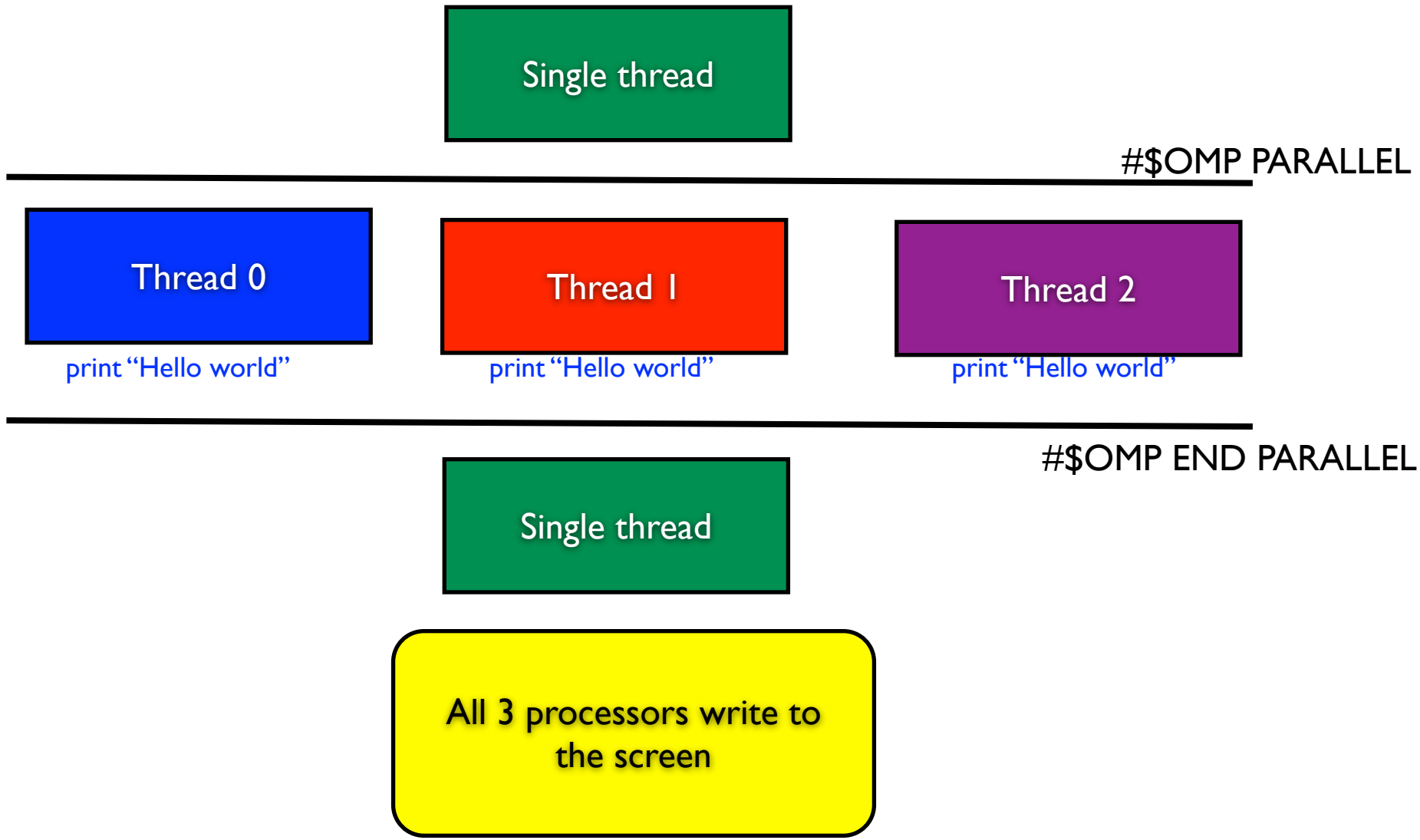
```
#include<stdio.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,”Hello world”);
}
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world
Hello world
Hello world
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world
Hello world
Hello world
```

Compile with openmp extensions

What is going on



Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
!$OMP PARALLEL
write(0,*) "Hello world",omp_get_thread_num(),"of",&
  omp_get_num_threads()
!$OMP END PARALLEL
end program
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,"Hello world %d of %d\n",
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
!$OMP PARALLEL
write(0,*) "Hello world",omp_get_thread_num(),"of",&
  omp_get_num_threads()
!$OMP END PARALLEL
end program
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,"Hello world %d of %d\n",
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

Note external functions in Fortran90

Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
!$OMP PARALLEL
write(0,*) “Hello world”,omp_get_thread_num(),“of”,&
  omp_get_num_threads()
!$OMP END PARALLEL
end program
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,”Hello world %d of %d\n”,
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

Function prototypes definitions

Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
!$OMP PARALLEL
write(0,*) “Hello world”,omp_get_thread_num(),“of”,&
  omp_get_num_threads()
!$OMP END PARALLEL
end program
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,”Hello world %d of %d\n”,
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

Return what thread the I am.

Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
!$OMP PARALLEL
write(0,*) "Hello world",omp_get_thread_num(),"of",&
  omp_get_num_threads()
!$OMP END PARALLEL
end program
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,"Hello world %d of %d\n",
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

The number of threads in the parallel environment.

Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
!$OMP PARALLEL
write(0,*) "Hello world",omp_get_thread_num(),"of",&
  omp_get_num_threads()
!$OMP END PARALLEL
end program
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,"Hello world %d of %d\n",
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

First thread is #0

Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
!$OMP PARALLEL
write(0,*) "Hello world",omp_get_thread_num(),"of",&
  omp_get_num_threads()
!$OMP END PARALLEL
end program
```

```
>setenv OMP_NUM_THREADS 3
>ifort -openmp hello.f90
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
 #pragama omp parallel
  fprintf(stderr,"Hello world %d of %d\n",
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>setenv OMP_NUM_THREADS 3
>icc -openmp hello.c
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

Print statements not necessarily in order or consistent.

Hello world

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
  int impi,nmpi;
 #pragama omp parallel
  {
   ith=omp_get_thread_num();
   nth=omp_get_num_threads();
  }
  fprintf(stderr,"Hello world %d of %d\n",
    ith,nth));
}
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
  int impi,nmpi;
 #pragama omp parallel
  {
   ith=omp_get_thread_num();
   nth=omp_get_num_threads();
   fprintf(stderr,"Hello world %d of %d\n",
    ith,nth));
  }
}
```

Hello world

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
  int impi,nmpi;
 #pragama omp parallel
  {
   ith=omp_get_thread_num();
   nth=omp_get_num_threads();
  }
  fprintf(stderr,"Hello world %d of %d\n",
    ith,nth));
}
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
  int impi,nmpi;
 #pragama omp parallel
  {
   ith=omp_get_thread_num();
   nth=omp_get_num_threads();
   fprintf(stderr,"Hello world %d of %d\n",
    ith,nth));
  }
}
```

Hello world

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
  int ith,nth;
 #pragama omp parallel
  {
   ith=omp_get_thread_num();
   nth=omp_get_num_threads();
  }
  fprintf(stderr,"Hello world %d of %d\n",
    ith,nth));
}
```

```
>./a.out
Hello world 1 of 3
>./a.out
Hello world 2 of 3
>./a.out
Hello world 1 of 3
```

Result not neccessarily consistent

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
  int ith,nth;
 #pragama omp parallel
  {
   ith=omp_get_thread_num();
   nth=omp_get_num_threads();
}
   fprintf(stderr,"Hello world %d of %d\n",
    ith,nth));
  }
```

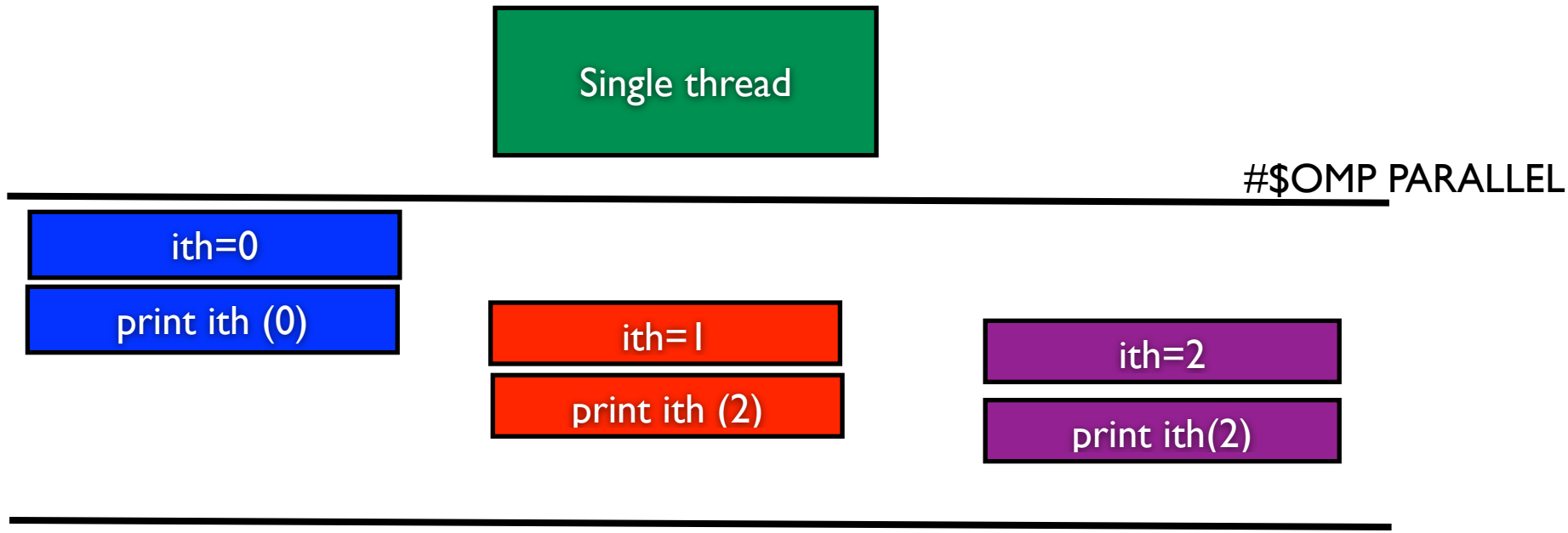
Hello world

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
  int impi,nmpi;
 #pragama omp parallel
  {
   ith=omp_get_thread_num();
   nth=omp_get_num_threads();
  }
  fprintf(stderr,"Hello world %d of %d\n",
    ith,nth));
}
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
  int impi,nmpi;
 #pragama omp parallel
  {
   ith=omp_get_thread_num();
   nth=omp_get_num_threads();
   fprintf(stderr,"Hello world %d of %d\n",
    ith,nth));
  }
}
```

```
>./a.out
Hello world 1 of 3
Hello world 2 of 3
Hello world 2 of 3
```

What is going on



Memory location can be changed between setting ith and printing ith

Public vs private

Within a parallel region every variable is assigned either accessible variable is assigned either a public or private status.

Public variables

- All threads access the same memory location (transfered to the heap)
- Thread timing is unpredictable so changes to the variable are unpredictable and inconsistent
- With some caveats thats that we will discuss later
- All things allocated on the stack must be public

Heap

- func()
  - a=b+c

Private variables

- Each thread has its own copy
- **Private variables exist on the stack**
  - stack requirements increase by nthreads
  - by default they disapear when exiting a parallel region

Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
integer :: ith,nth
!$OMP PARALLEL default(shared) private(ith,nth)
ith=omp_get_thread_num()
nth=omp_get_num_threads()
write(0,*) “Hello world”,ith,nth
!$OMP END PARALLEL
end program
```

```
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
int ith, nth;
 #pragama omp parallel default(shared) private(ith,nth)
{
 ith=omp_get_thread_num()
   nth=omp_get_num_threads()
  fprintf(stderr,”Hello world %d of %d\n”,
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
integer :: ith,nth
!$OMP PARALLEL default(shared) private(ith,nth)
ith=omp_get_thread_num()
nth=omp_get_num_threads()
write(0,*) “Hello world”,ith,nth
!$OMP END PARALLEL
end program
```

```
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
int ith, nth;
 #pragama omp parallel default(shared) private(ith,nth)
   ith=omp_get_thread_num()
   nth=omp_get_num_threads()
  fprintf(stderr,”Hello world %d of %d\n”,
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

By default all variables are shared

Hello world

```
program hello_world
integer, external :: omp_get_thread_num
integer, external :: omp_get_num_threads
integer :: ith,nth
!$OMP PARALLEL default(shared) private(ith,nth)
ith=omp_get_thread_num()
nth=omp_get_num_threads()
write(0,*) “Hello world”,ith,nth
!$OMP END PARALLEL
end program
```

```
>./a.out
Hello world    1 of    3
Hello world    2 of    3
Hello world    0 of    3
```

```
#include<stdio.h>
#include<omp.h>
int main (int argc, char **argv)
{
int ith, nth;
 #pragama omp parallel default(shared) private(ith,nth)
   ith=omp_get_thread_num()
   nth=omp_get_num_threads()
  fprintf(stderr,”Hello world %d of %d\n”,
    omp_get_thread_num(), omp_get_num_threads());
}
```

```
>./a.out
Hello world 0 of 3
Hello world 2 of 3
Hello world 1 of 3
```

The list of variables that are private

OMP do/for construct

- A loop is executed by multiple threads
- Data parallel approach

Vector multiplication

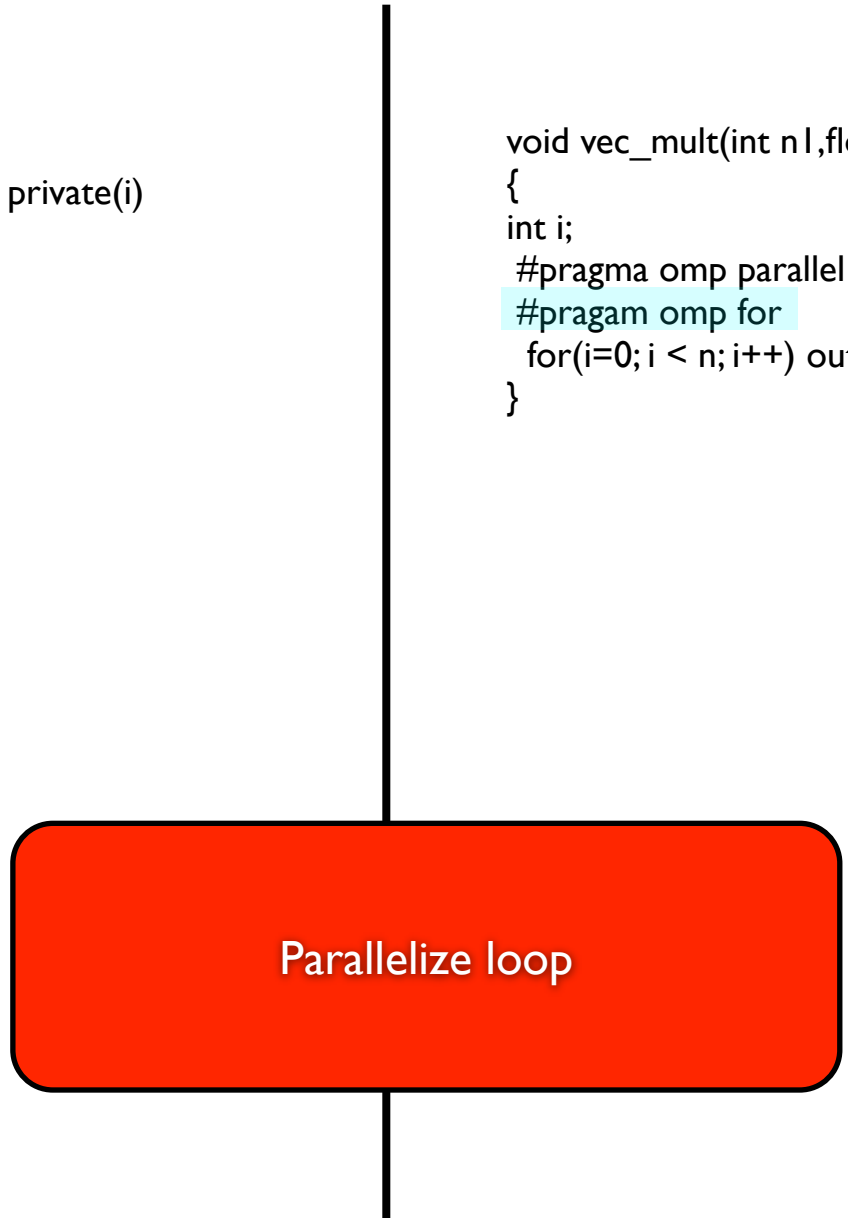
```
subroutine vec_mult(out,in1,in2)
  real :: out(:).in1(:),in2(:)
  integer :: i
!$OMP PARALLEL default(shared) private(i)
!$OMP DO
do i=1,size(out)
  out(i)=in1(i)*in2(i)
end do
!$OMP END DO
!$OMP END PARALLEL
end subroutine
```

```
void vec_mult(int n1,float *out,float *in1,float *in2)
{
int i;
 #pragma omp parallel default(shared) private(i)
 #pragam omp for
  for(i=0; i < n; i++) out[i]=in1[i]*in2[i];
}
```

Vector multiplication

```
subroutine vec_mult(out,in1,in2)
  real :: out(:).in1(:),in2(:)
  integer :: i
!$OMP PARALLEL default(shared) private(i)
!$OMP DO
do i=1,size(out)
  out(i)=in1(i)*in2(i)
end do
!$OMP END DO
!$OMP END PARALLEL
end subroutine
```

```
void vec_mult(int n1,float *out,float *in1,float *in2)
{
int i;
 #pragma omp parallel default(shared) private(i)
 #pragam omp for
  for(i=0; i < n; i++) out[i]=in1[i]*in2[i];
}
```



Vector multiplication

```
subroutine vec_mult(out,in1,in2)
  real :: out(:).in1(:),in2(:)
  integer :: i
!$OMP PARALLEL DO default(shared) private(i)
do i=1,size(out)
  out(i)=in1(i)*in2(i)
end do
!$OMP END PARALLEL DO
end subroutine
```

```
void vec_mult(int n1,float *out,float *in1,float *in2)
{
int i;
 #pragama omp parallel for default(shared) private(i)
  for(i=0; i < n; i++) out[i]=in1[i]*in2[i];
}
```

Loop and parallel compiler directives can be combined.

Dot product (version 1)

```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  integer :: i
  dot_product=0
!$OMP PARALLEL DO default(shared) private(i)
do i=1,size(in1)
  dot_product=dot_product+in1(i)*in2(i)
end do
!$OMP END PARALLEL DO
end function
```

```
double dot_product(int n, float *in1, *in2)
{
  int i;
  double out=0.;
  #pragma omp parallel for default(shared) private(i)
   for(i=0; i < n; i++)out+=in1[i]*in2[i];
  return(out);
}
```

Dot product (version 1)

```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  integer :: i
  dot_product=0
!$OMP PARALLEL DO default(shared) private(i)
do i=1,size(in1)
  dot_product=dot_product+in1(i)*in2(i)
end do
!$OMP END PARALLEL DO
end function
```

```
double dot_product(int n, float *in1, *in2)
{
  int i;
  double out=0.;
  #pragma omp parallel for default(shared) private(i)
  for(i=0; i < n; i++)out+=in1[i]*in2[i];
  return(out);
}
```

Race condition (different threads may not send to heap updated out quickly enough)

Dot product (version 1)

```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  integer :: i
  dot_product=0
!$OMP PARALLEL DO default(shared) private(i) &
do i=1,size(in1)
  !$OMP ATOMIC
  dot_product=dot_product+in1(i)*in2(i)
end do
!$OMP END PARALLEL DO
end function
```

```
double dot_product(int n, float *in1, *in2)
{
  int i;
  double out=0.;
  #pragma omp parallel for default(shared) private(i)
   {
   for(i=0; i < n; i++){
     #pragma omp atomic
     out+=in1[i]*in2[i];
    }
  }
   return(out);
}
```

Guarantees that statement will be completed and memory location updated before another processors accesses it.
Limited number of constructs (+=,-=, etc.)

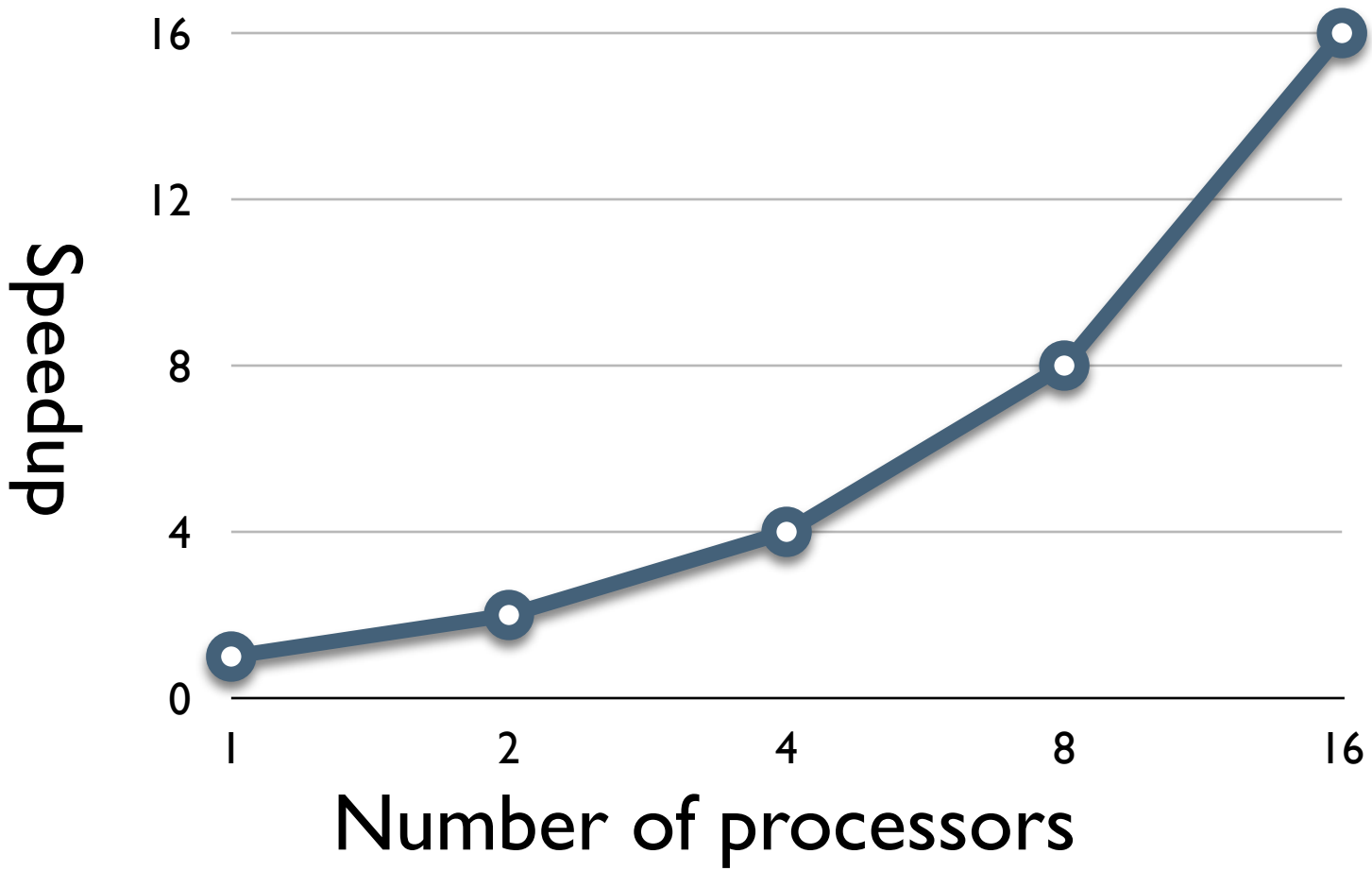
Dot product (version 1)

```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  integer :: i
  dot_product=0
!$OMP PARALLEL DO default(shared) private(i)
do i=1,size(in1)
   !$OMP CRITICAL
  dot_product=dot_product+in1(i)*in2(i)
!$OMP END CRITICAL
end do
   !$OMP END CRITICAL
!$OMP END PARALLEL DO
end function
```

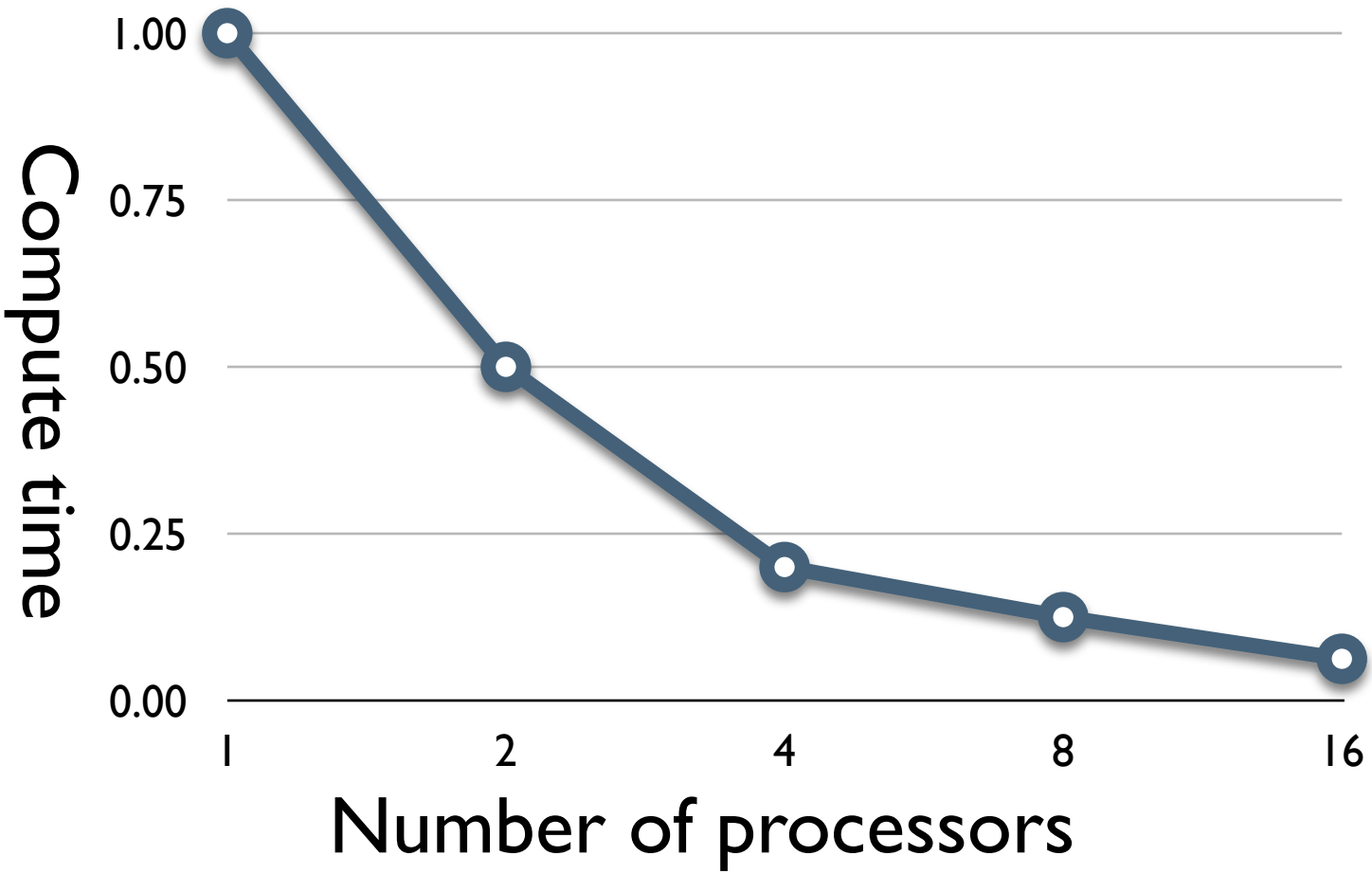
```
double dot_product(int n, float *in1, *in2)
{
  int i;
  double out=0.;
  #pragma omp parallel for default(shared) private(i)
   {
   for(i=0; i < n; i++){
      #pragma omp critical
      out+=in1[i]*in2[i];
    }
  }
   return(out);
}
```

Guarantees that only one thread at a time will enter the code section.

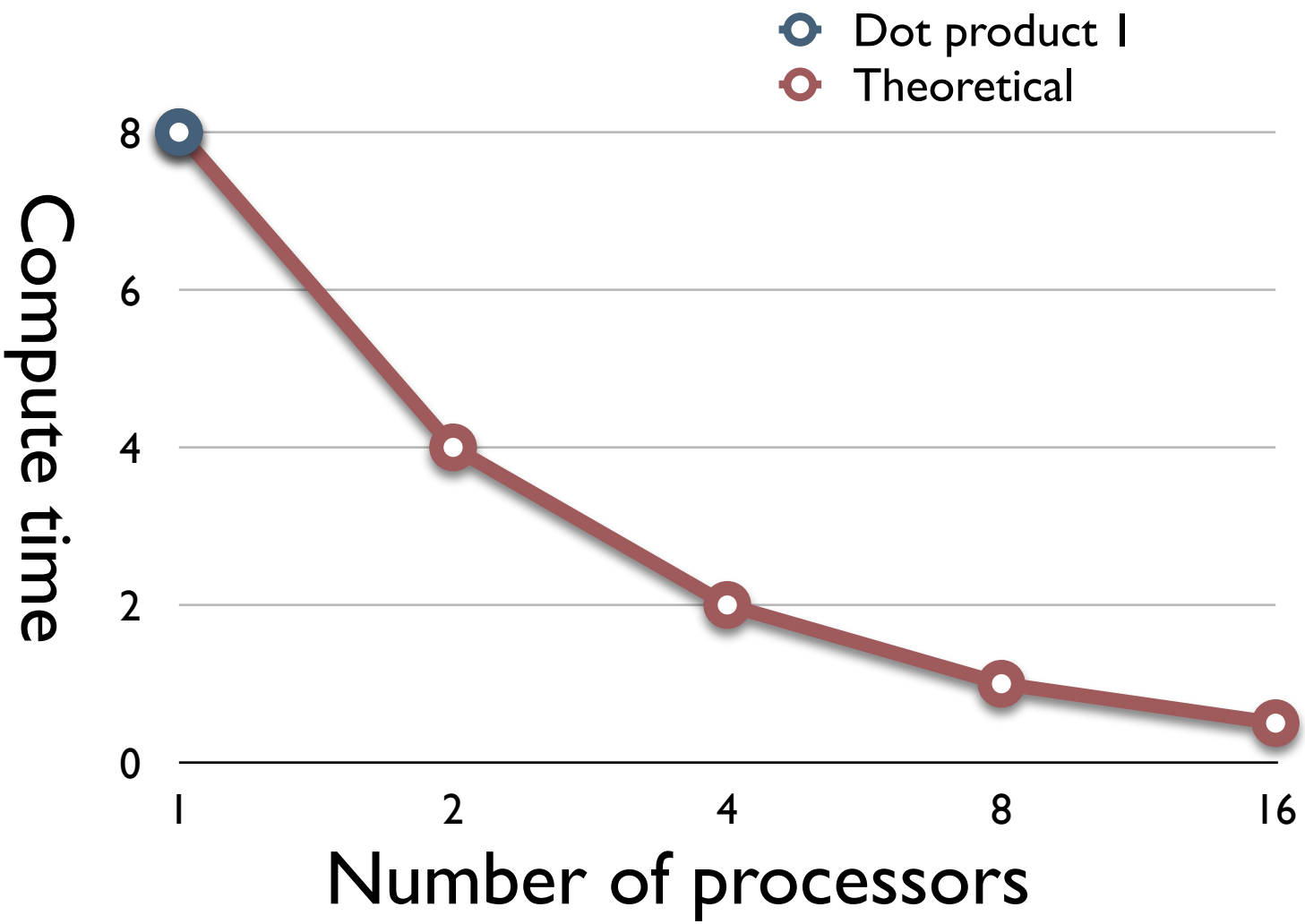
Theoretical speedup



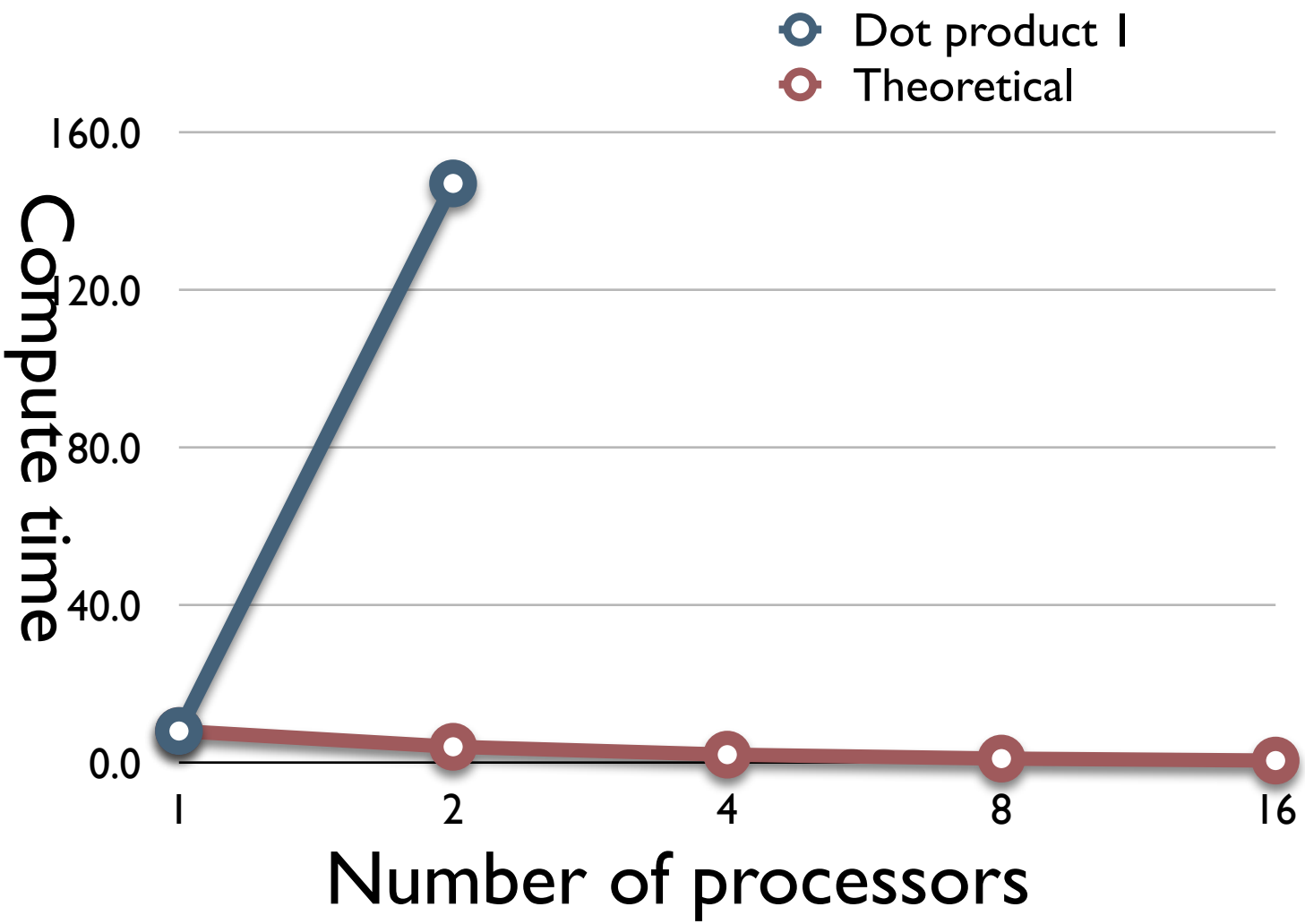
Compute time



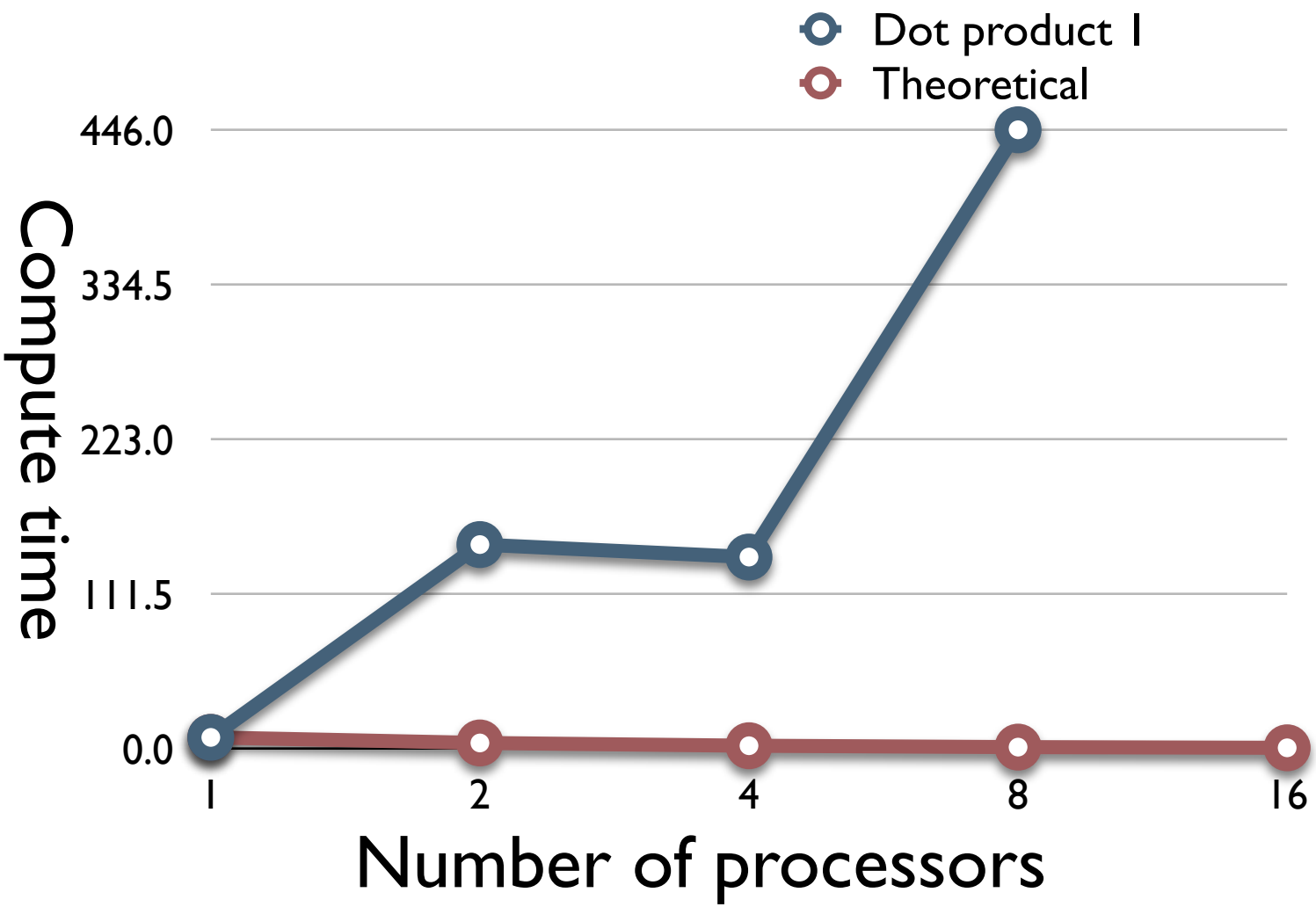
Dot product(1)



Dot product(1)



Dot product(1)



What is going on

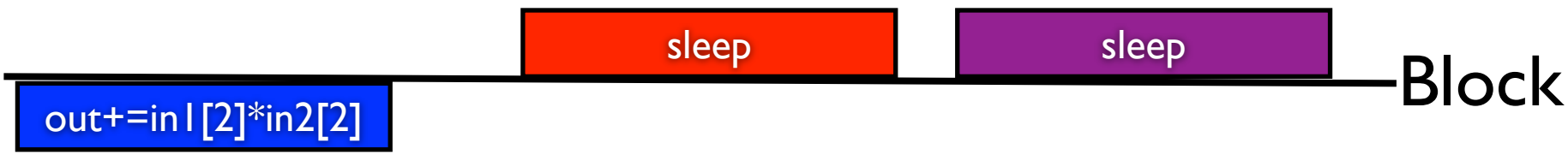
out+=in1[0]*in2[0]

First thread creates block

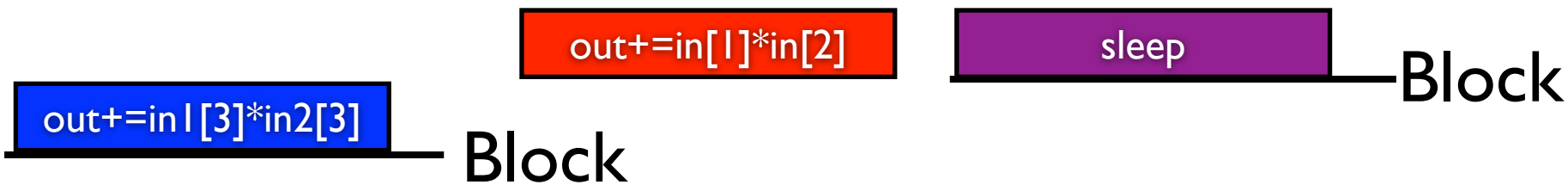
Other thread hits block

Other thread check status

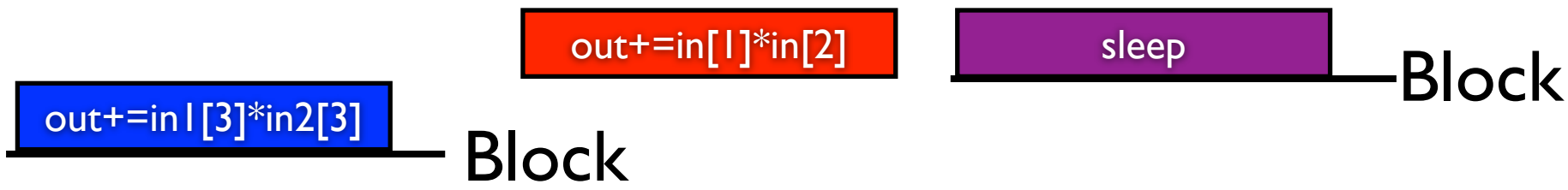
Other threads sleep



Next thread is unblocked

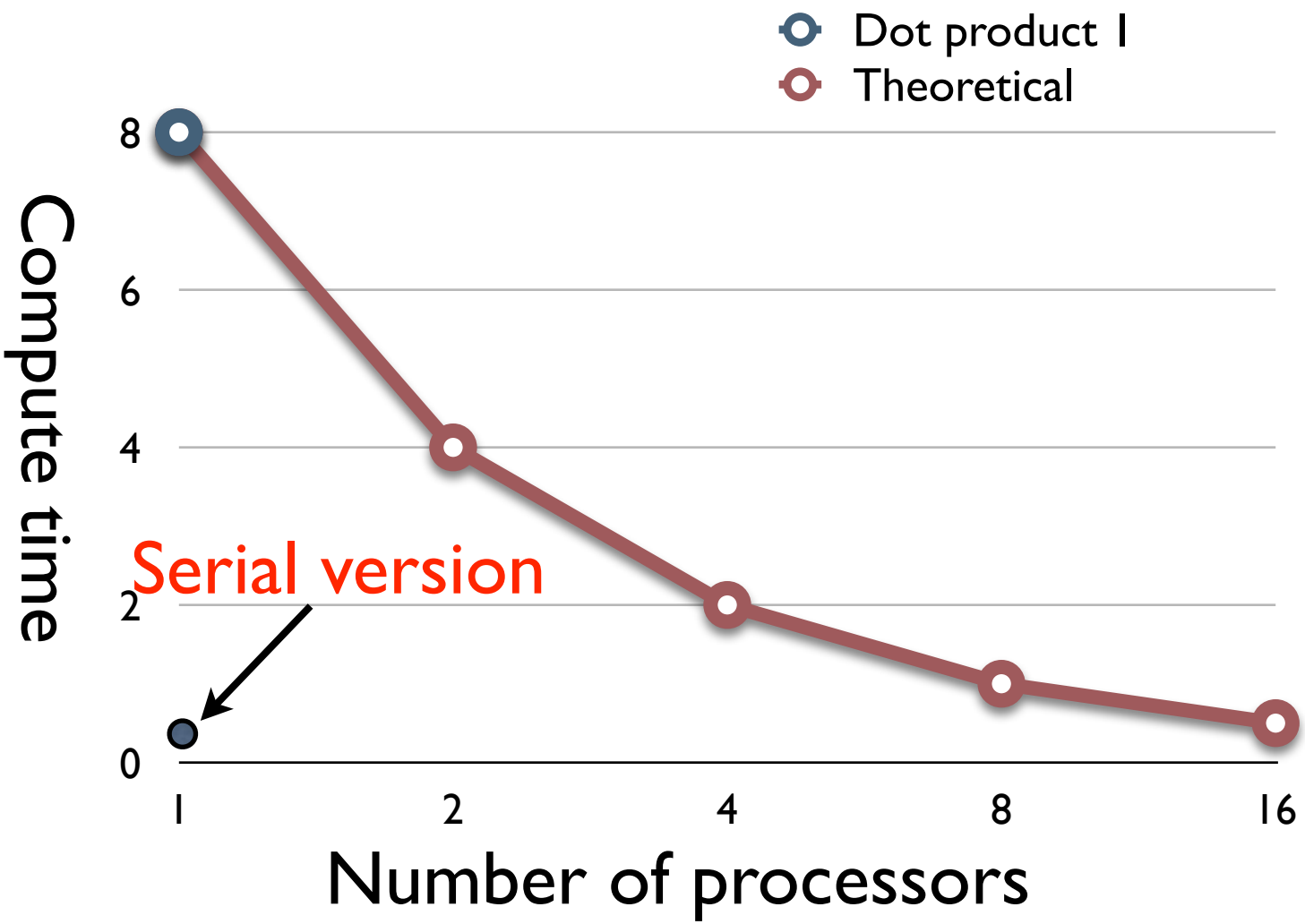


Next thread hits block

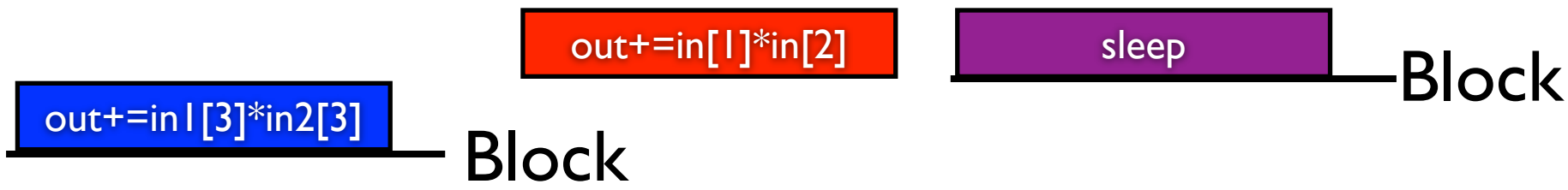


Lock checking/sleep combo is a non-trivial cost

Dot product(1)



Next thread hits block



The cost of setting up the locks 20x the cost of doing the addition

Dot product (version 2)

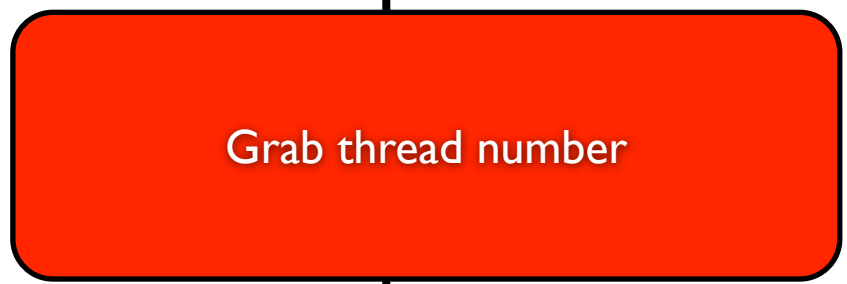
```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  double precision :: dots(100)
  integer :: i,ith
  integer,external :: omp_get_thread_num
  dots=0;dot_product=0
!$OMP PARALLEL default(shared) private(ith,i)
ith=omp_get_thread_num()
!$OMP DO
do i=1,size(in1)
  dots(ith+1)=dots(ith+1)+in1(i)*in2(i)
 end do
!$OMP END DO
!$OMP END PARALLEL
dot_product=sum(dots)
end function
```

```
double dot_product(int n, float *in1, *in2)
{
  int i,ith;
  double dots[100], out=0.;
  for(i=0; i < 100; i++) dots[i]=0.;
  #pragma omp parallel default(shared) private(ith,i)
  ith=omp_get_thread_num();
  #pragma omp for
   {
   for(i=0; i < n; i++)i{
       #pragma omp critical
       dots[ith]+=in1[i]*in2[i];
    }
  }
   for(i=0; i < 100; i++) out+=dots[i];
   return(out);
}
```

Dot product (version 2)

```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  double precision :: dots(100)
  integer :: i,ith
  integer,external :: omp_get_thread_num
  dots=0;dot_product=0
!$OMP PARALLEL default(shared) private(i,ith)
ith=omp_get_thread_num()
!$OMP DO
do i=1,size(in1)
  dots(ith+1)=dots(ith+1)+in1(i)*in2(i)
 end do
!$OMP END DO
!$OMP END PARALLEL
dot_product=sum(dots)
end function
```

```
double dot_product(int n, float *in1, *in2)
{
  int i,ith;
  double dots[100], out=0.;
  for(i=0; i < 100; i++) dots[i]=0.;
  #pragma omp parallel default(shared) private(ith,i)
  ith=omp_get_thread_num();
  #pragma omp for
   {
   for(i=0; i < n; i++)i{
       dots[ith]+=in1[i]*in2[i];
    }
  }
   for(i=0; i < 100; i++) out+=dots[i];
   return(out);
}
```



Dot product (version 2)

```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  double precision :: dots(100)
  integer :: i,ith
  integer,external :: omp_get_thread_num
  dots=0;dot_product=0
!$OMP PARALLEL default(shared) private(i,ith)
ith=omp_get_thread_num()
!$OMP DO
do i=1,size(in1)
  dots(ith+1)=dots(ith+1)+in1(i)*in2(i)
 end do
!$OMP END DO
!$OMP END PARALLEL
dot_product=sum(dots)
end function
```

```
double dot_product(int n, float *in1, *in2)
{
  int i,ith;
  double dots[100], out=0.;
  for(i=0; i < 100; i++) dots[i]=0.;
  #pragma omp parallel default(shared) private(ith,i)
  ith=omp_get_thread_num();
  #pragma omp for
   {
   for(i=0; i < n; i++)i{
      dots[ith]+=in1[i]*in2[i];
    }
  }
   for(i=0; i < 100; i++) out+=dots[i];
   return(out);
}
```

Local dot product sum for each thread

Dot product (version 2)

```
double precision  function dot_product(in1,in2)
  real :: in1(:),in2(:)
  double precision :: dots(100)
  integer :: i,ith
  integer,external :: omp_get_thread_num
  dots=0;dot_product=0
!$OMP PARALLEL default(shared) private(ith,i)
ith=omp_get_thread_num()
!$OMP DO
do i=1,size(in1)
  dots(ith+1)=dots(ith+1)+in1(i)*in2(i)
 end do
!$OMP END DO
!$OMP END PARALLEL
dot_product=sum(dots)
end function
```

```
double dot_product(int n, float *in1, *in2)
{
  int i,ith;
  double dots[100], out=0.;
  for(i=0; i < 100; i++) dots[i]=0.;
  #pragma omp parallel default(shared) private(ith,i)
  ith=omp_get_thread_num();
  #pragma omp for
   {
   for(i=0; i < n; i++)i{
       dots[ith]+=in1[i]*in2[i];
    }
  }
   for(i=0; i < 100; i++) out+=dots[i];
   return(out);
}
```

Combine local dot product results.

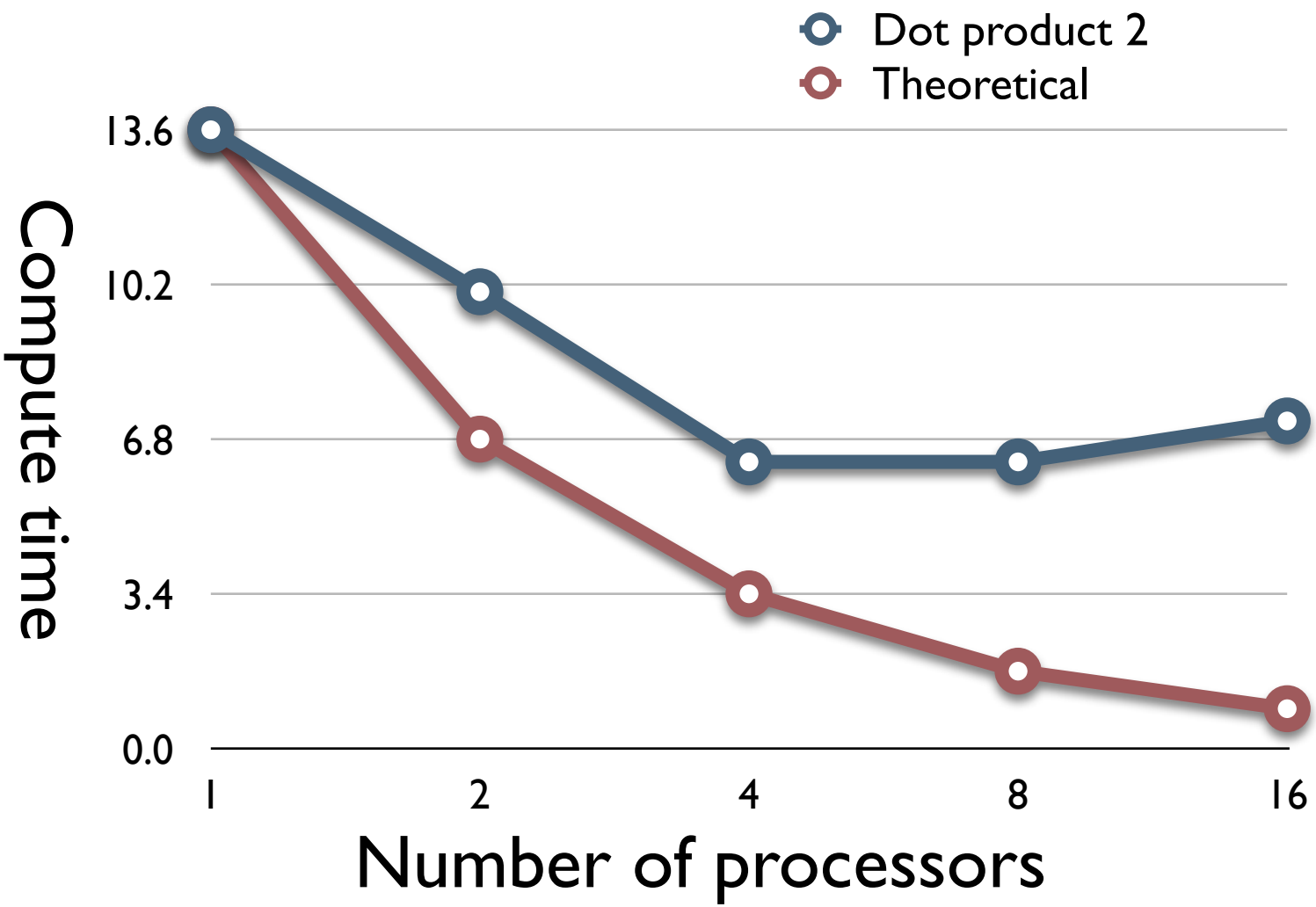
Dot product (version 2)

```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  double precision :: dots(100)
  integer :: i,ith
  integer,external :: omp_get_thread_num
  dots=0;dot_product=0
!$OMP PARALLEL default(shared) private(ith,i)
ith=omp_get_thread_num()
!$OMP DO
do i=1,size(in1)
  dots(ith+1)=dots(ith+1)+in1(i)*in2(i)
 end do
!$OMP END DO
!$OMP END PARALLEL
dot_product=sum(dots)
end function
```

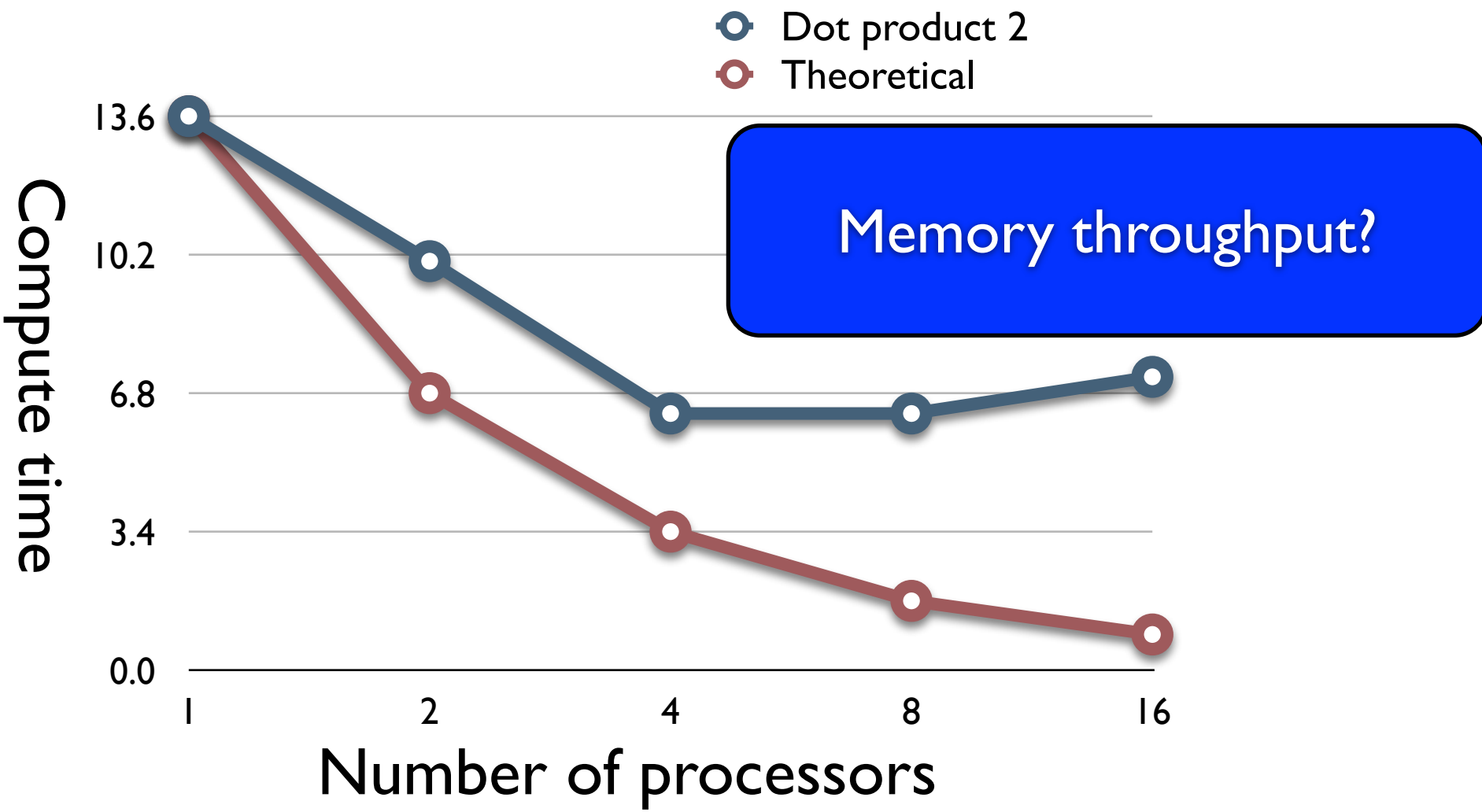
```
double dot_product(int n, float *in1, *in2)
{
  int i,ith;
  double dots[100], out=0.;
  for(i=0; i < 100; i++) dots[i]=0.;
  #pragma omp parallel default(shared) private(ith,i)
  ith=omp_get_thread_num();
  #pragma omp for
   {
   for(i=0; i < n; i++)i{
       dots[ith]+=in1[i]*in2[i];
    }
  }
   for(i=0; i < 100; i++) out+=dots[i];
   return(out);
}
```

Remove all blocking from the loop.

Dot product(2)



Dot product(2)



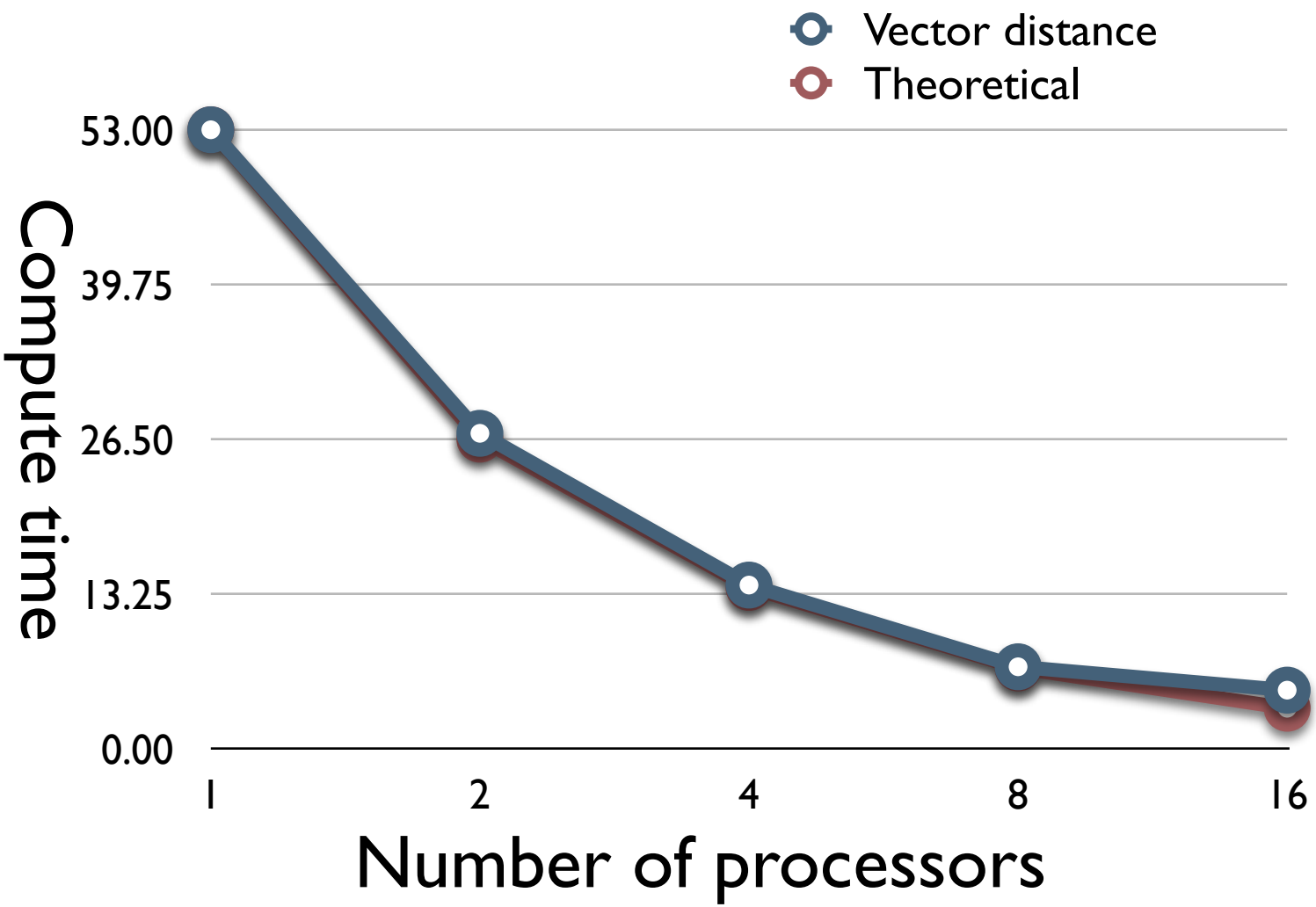
Vector distance

```
double precision function vec_dist(in1,in2)
  real :: in1(:),in2(:)
  double precision :: dots(100)
  integer :: i,ith
  integer,external :: omp_get_thread_num
  dots=0;dot_product=0
!$OMP PARALLEL default(shared) private(ith,i)
ith=omp_get_thread_num()
!$OMP DO
do i=1,size(in1)
  dots(ith+1)=dots(ith+1)+sqrt(in1(i)**2+in2(i)**2)
 end do
!$OMP END DO
!$OMP END PARALLEL
vec_dist=sum(dots)
end function
```

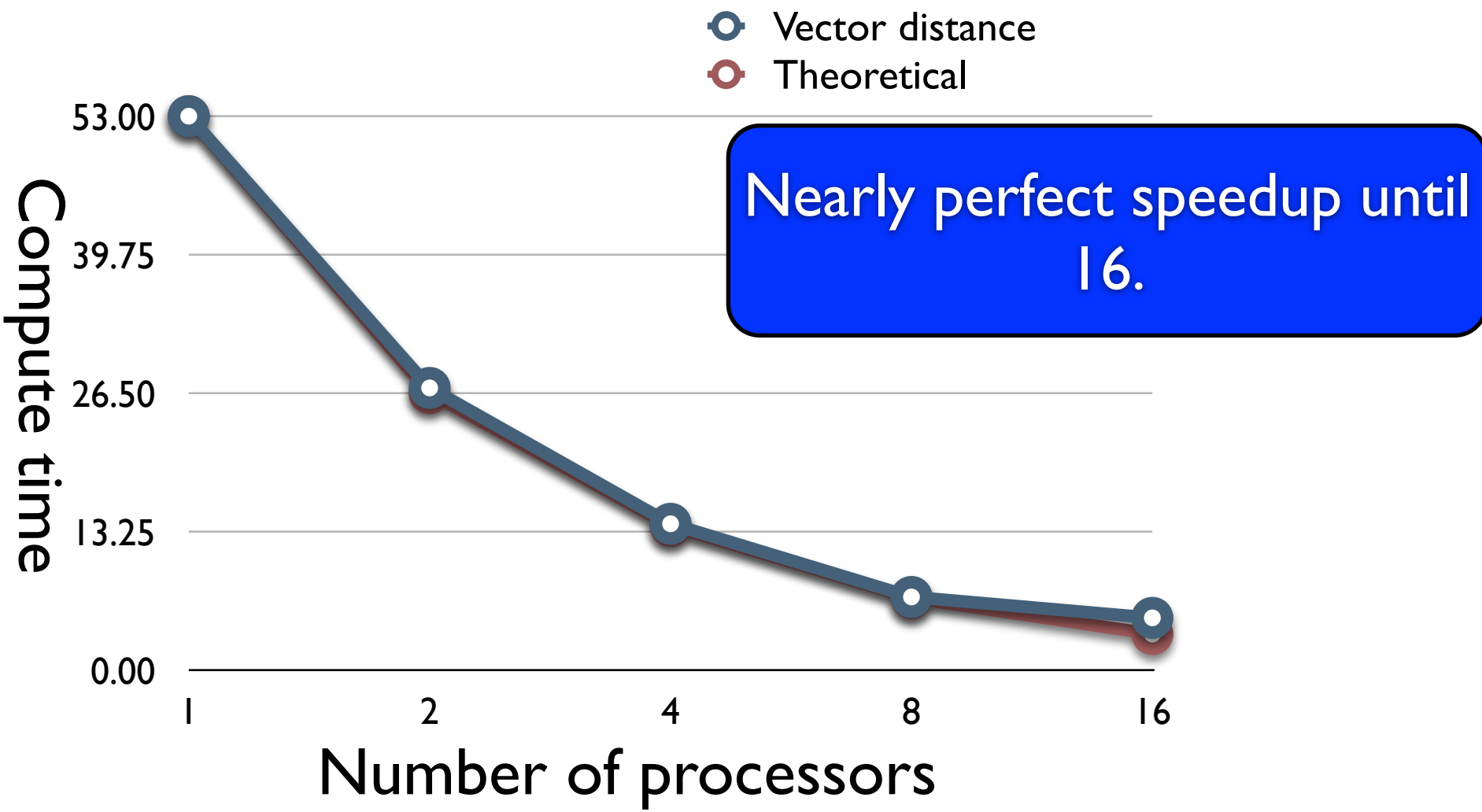
```
double vec_dist(int n, float *in1, *in2)
{
  int i,ith;
  double dots[100], out=0.;
  for(i=0; i < 100; i++) dots[i]=0.;
  #pragma omp parallel default(shared) private(ith,i)
  ith=omp_get_thread_num();
  #pragma omp for
   {
   for(i=0; i < n; i++)i{
       dots[ith]+=sqrt(in1[i]*in1[i]+in2[i]*in2[i]);
    }
  }
   for(i=0; i < 100; i++) out+=dots[i];
   return(out);
}
```

More operations per datapoint

Vector distance



Vector distance



OpenMP scheduling

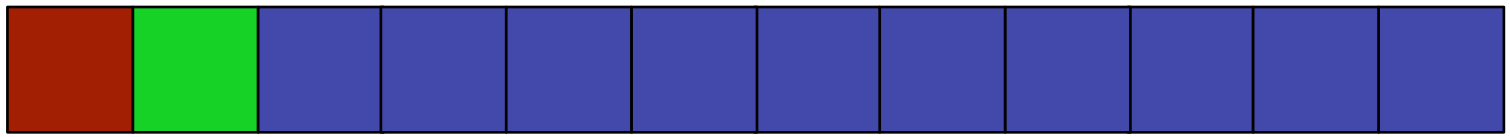
- Static - Broken into equal blocks (low overhead
- Dynamic - Once a thread finishes it is assigned next available loop iteration
- Other scheduling (affinity, guided, runtime)

Scheduling: Static (2 threads)

Thread 1

Thread 2

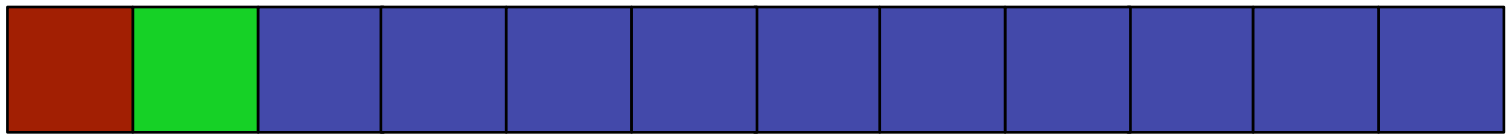
Scheduling: Dynamic(2 threads)



Thread 1

Thread 2

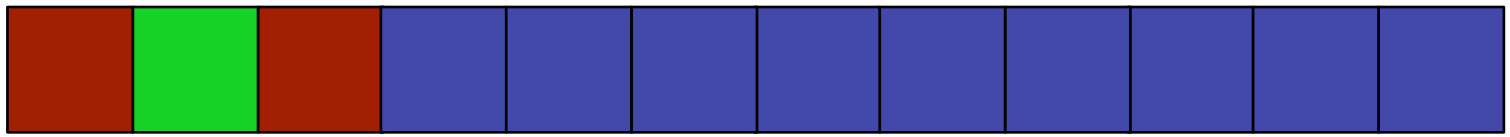
Scheduling: Dynamic(2 threads)



Thread 1

Thread 2

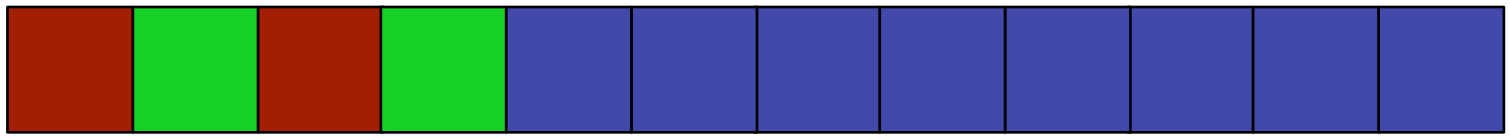
Scheduling: Dynamic(2 threads)



Thread 1

Thread 2

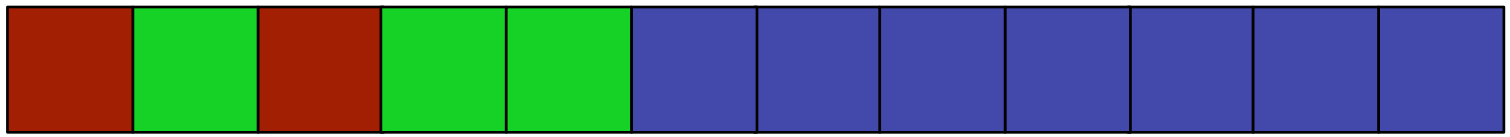
Scheduling: Dynamic(2 threads)



Thread 1

Thread 2

Scheduling: Dynamic(2 threads)



Thread 1

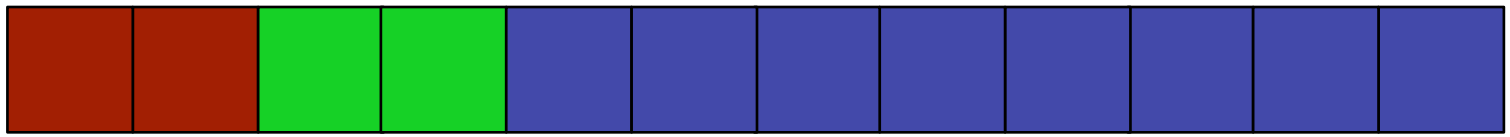
Thread 2

Scheduling: Dynamic (2 threads)

Thread 1

Thread 2

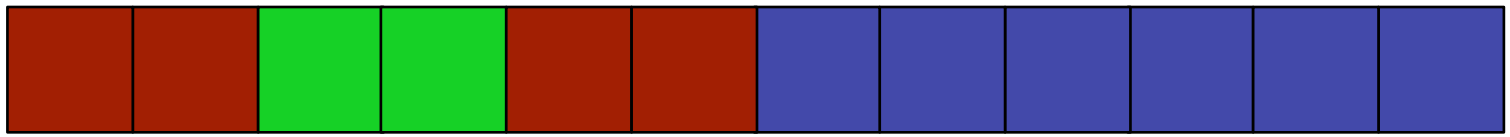
Scheduling: Dynamic specifying a block size (2 threads)



Thread 1

Thread 2

Scheduling: Dynamic specifying a block size (2 threads)



Thread 1

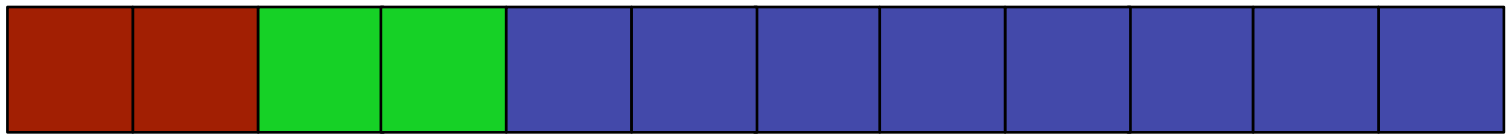
Thread 2

Scheduling: Dynamic specifying a block size (2 threads)

Thread 1

Thread 2

Scheduling: Dynamic specifying a block size (2 threads)



Thread 1

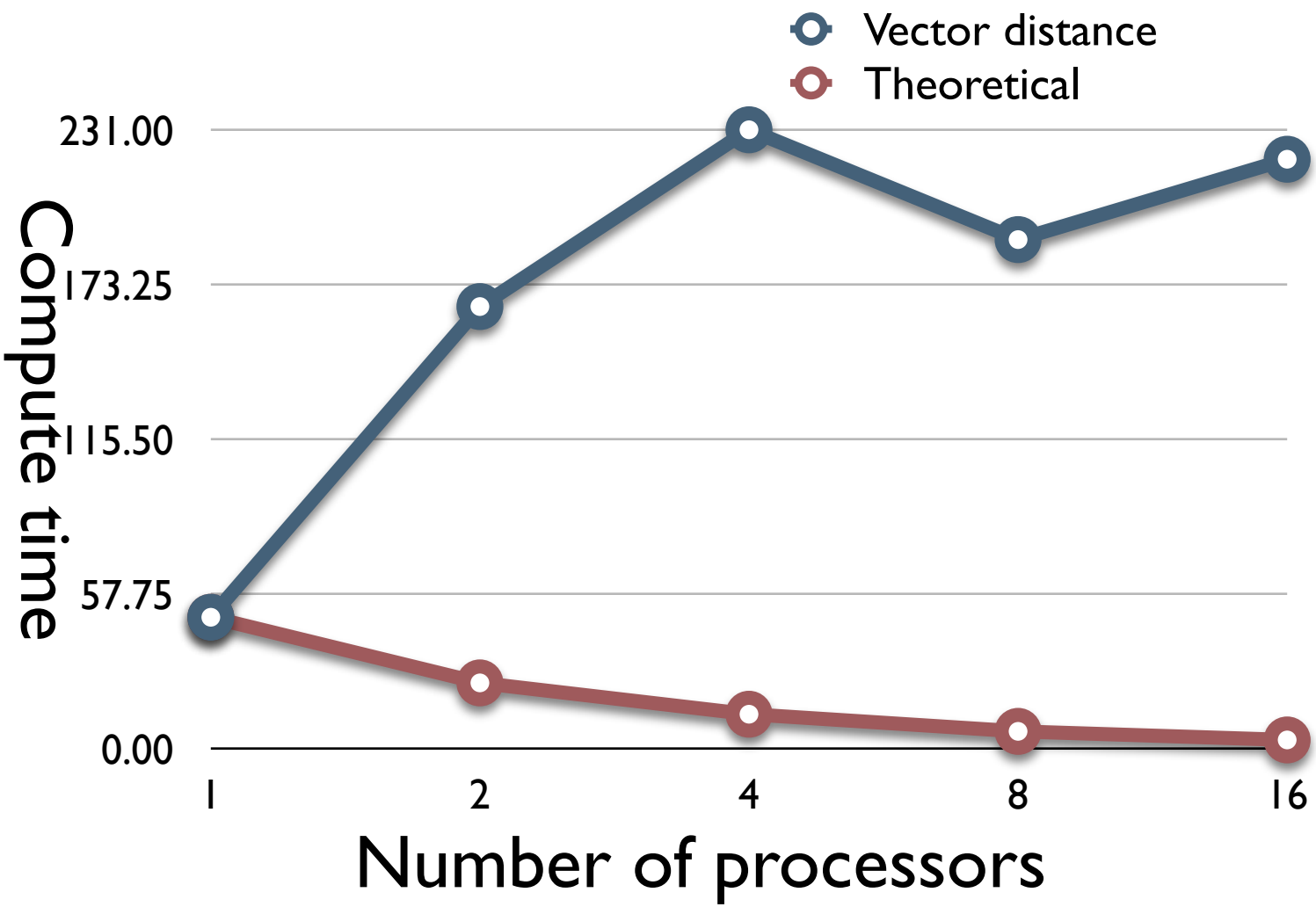
Thread 2

Vector distance

```
double precision function vec_dist(in1,in2)
  real :: in1(:),in2(:)
  double precision :: dots(100)
  integer :: i,ith
  integer,external :: omp_get_thread_num
  dots=0;dot_product=0
!$OMP PARALLEL default(shared) private(i,ith)
ith=omp_get_thread_num()
!$OMP DO schedule(dynamic,5)
do i=1,size(in1)
  dots(ith+1)=dots(ith+1)+sqrt(in1(i)**2+in2(i)**2)
 end do
!$OMP END DO
!$OMP END PARALLEL
vec_dist=sum(dots)
end function
```

```
double vec_dist(int n, float *in1, *in2)
{
  int i,ith;
  double dots[100], out=0.;
  for(i=0; i < 100; i++) dots[i]=0.;
  #pragma omp parallel default(shared) private(ith,i)
  ith=omp_get_thread_num();
  #pragma omp for schedule(dynamic,5)
   {
   for(i=0; i < n; i++)i{
       dots[ith]+=sqrt(in1[i]*in1[i]+in2[i]*in2[i]);
    }
  }
   for(i=0; i < 100; i++) out+=dots[i];
   return(out);
}
```

Vector distance: dynamic



Dot product (version 3)

```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  integer :: i,ith
  integer,external :: omp_get_thread_num
  dots=0;dot_product=0
!$OMP PARALLEL default(shared) private(i,ith)
ith=omp_get_thread_num()
!$OMP REDUCTION(+:dot_product)
!$OMP DO
do i=1,size(in1)
  dot_product=dots(ith+1)+in1(i)*in2(i)
 end do
!$OMP END DO
!$OMP END PARALLEL
end function
```

```
double dot_product(int n, float *in1, *in2)
{
  int i,ith;
  double  out=0.;
  #pragma omp parallel default(shared) private(ith,i)
     reduction(+:out)
  ith=omp_get_thread_num();
  #pragma omp for
   {
   for(i=0; i < n; i++)i{
       out+=in1[i]*in2[i];
    }
  }
   return(out);
}
```

Dot product (version 3)

```
double precision function dot_product(in1,in2)
  real :: in1(:),in2(:)
  integer :: i,ith
  integer,external :: omp_get_thread_num
  dots=0;dot_product=0
!$OMP PARALLEL default(shared) private(i)
!$OMP REDUCTION(+:dot_product)
ith=omp_get_thread_num()
!$OMP DO
do i=1,size(in1)
  dot_product=dots(ith+1)+in1(i)*in2(i)
 end do
!$OMP END DO
!$OMP END PARALLEL
end function
```

```
double dot_product(int n, float *in1, *in2)
{
  int i,ith;
  double  out=0.;
  #pragma omp parallel default(shared) private(ith,i)
      reduction(+:out)
  ith=omp_get_thread_num();
 #pragma omp for
   {
   for(i=0; i < n; i++)i{
       out+=in1[i]*in2[i];
    }
  }
   return(out);
}
```

All values are combined at the end of the parallel clause.

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

```
a=4
!$OMP PARALLEL DO default(shared) private(i)
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
     print i,a,b
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

```
a=4
!$OMP PARALLEL DO default(shared) private(i)
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
    print i,a,b
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Test: Four bugs/differences in this conversion

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

```
a=4
!$OMP PARALLEL DO default(shared) private(i)
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
    print i,a,b
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Problem 1:
a and b are clobered within
the do loop

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

```
a=4
!$OMP PARALLEL DO default(shared) private(i,a,b)
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
    print i,a,b
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Solution 1:
Make a and b private variables

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

```
a=4
!$OMP PARALLEL DO default(shared) private(i,a,b)
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
    print i,a,b
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Problem 2:
a is initialized to 0 when entering parallel environment

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

```
a=4
!$OMP PARALLEL DO default(shared) private(i,a,b)&
!$OMP & firstprivate(a)
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
    print i,a,b
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Solution 2:
firstprivate clause copies variable from serial portion to parallel region

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

```
a=4
!$OMP PARALLEL DO default(shared) private(i,a,b)&
!$OMP & firstprivate(a)
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
    print i,a,b
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Problem 3: b disapears when leaving the parallel environment.

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

```
a=4
!$OMP PARALLEL DO default(shared) private(i,a,b)&
!$OMP & firstprivate(a) lastprivate(b)
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
    print i,a,b
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Solution: lastprivate copies from the last parallel thread back to the serial code portion.

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

```
a=4
!$OMP PARALLEL DO default(shared) private(i,a,b)&
!$OMP & firstprivate(a) lastprivate(b)
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
    print i,a,b
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Problem: Print statements will not be in the same order

Serial to OpenMP gotchas

```
a=4
do i=1,n
  call big_stuff(array,i)
  a=a-1
  b=a+i
  print i,a,b
  call big_stuff2(array,i)
end do
print b
```

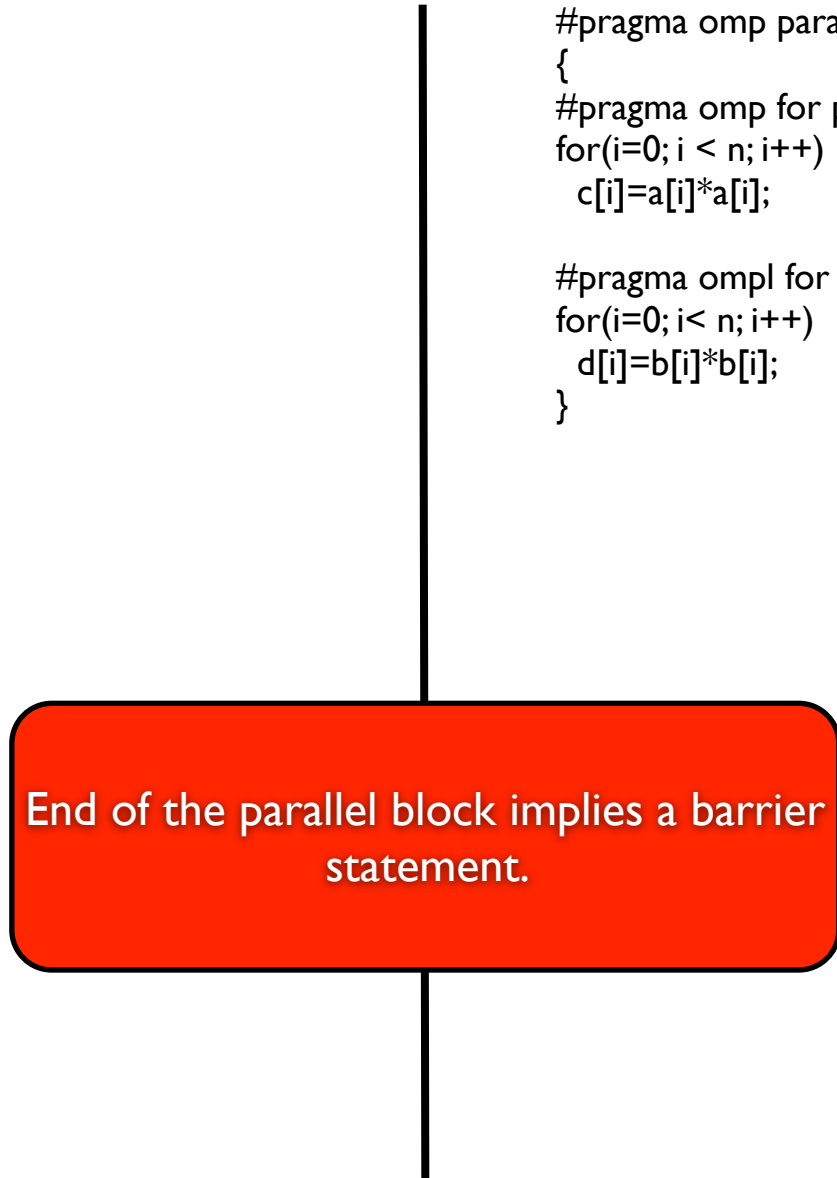
```
a=4
!$OMP PARALLEL DO default(shared) private(i,a,b)&
!$OMP & firstprivate(a) lastprivate(b) ordered
do i=1,n
   call big_stuff(array,i)
    a=a-1
    b=a+i
   !$OMP ORDERED
    print i,a,b
   !$OMP END ORDERED
    call big_stuff2(array,i)
end do
!$OMP END PARALLEL DO
print b
```

Solution: Ordered
Guarantees that code block will be executed in the same manner as serial code

OMP nowait

```
for(i=0; i < n; i++)
  c[i]=a[i]*a[i];

for(i=0; i< n; i++)
  d[i]=b[i]*b[i];
```

```
#pragma omp parallel
{
#pragma omp for private(i)
for(i=0; i < n; i++)
  c[i]=a[i]*a[i];

#pragma ompl for private(i)
for(i=0; i< n; i++)
  d[i]=b[i]*b[i];
}
```



OMP nowait

```
for(i=0; i < n; i++)
  c[i]=a[i]*a[i];

for(i=0; i< n; i++)
  d[i]=b[i]*b[i];
```

```
#pragma omp parallel nowait
{
#pragma omp for private(i)
for(i=0; i < n; i++)
  c[i]=a[i]*a[i];

#pragma ompl for private(i)
for(i=0; i< n; i++)
  d[i]=b[i]*b[i];
}
```

If loops aren't dependent the nowait clause removes the barrier statement.

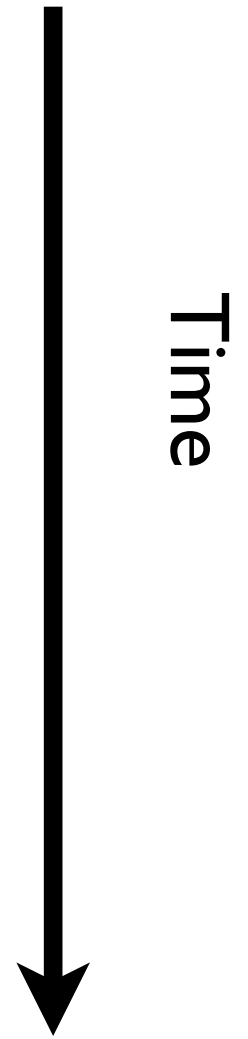
OMP workshare

```
real a(:),b(:),c(:)

c=b+a
```

```
real a(:),b(:),c(:)

#OMP PARALLEL WORKSHARE
c=b+a
#OMP END PARALLEL WORKSHARE
```

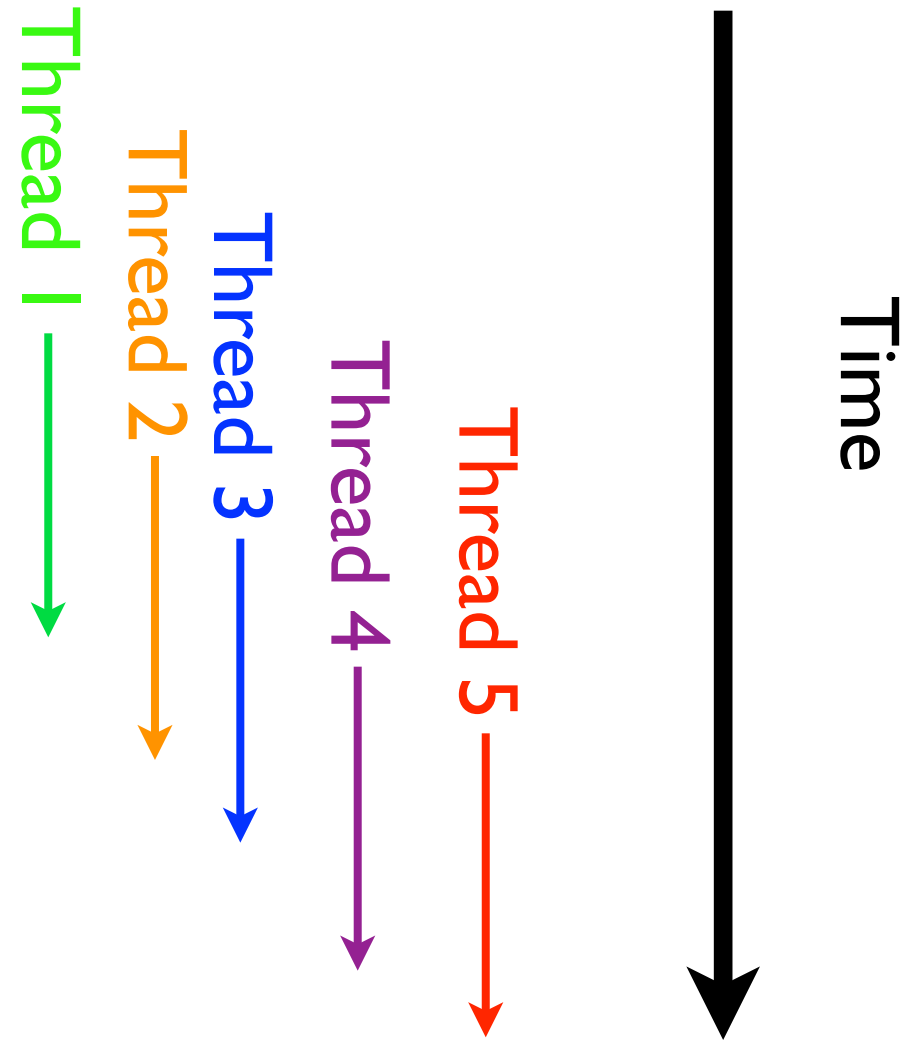
Threads model

```
program alpha
.
.
call sub1()
call sub2()
call sub3()
call sub4()
call sub5()
.
.
end program
```



Threads model

```
program alpha
.
.
call sub1()
call sub2()
call sub3()
call sub4()
call sub5()
.
.
end program
```



Sections

```
program alpha
.
.
!$OMP PARALLEL SECTIONS
call sub1()
call sub2()
call sub3()
call sub4()
call sub5()
!$END PARALLEL SECTIONS
.
end program
```

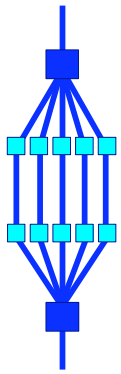
Begin and end functional parallel environment

Sections

```
program alpha
.
.
!$OMP PARALLEL SECTIONS
!$OMP SECTION
call sub1()
!$OMP END SECTION
!$OMP SECTION
call sub2()
!$OMP END SECTION
!$OMP SECTION
call sub3()
!$OMP END SECTION
!$OMP SECTION
call sub4()
!$OMP END SECTION
!$OMP SECTION
call sub5()
!$OMP END SECTION
!$END PARALLEL SECTIONS
.
end program
```

Only a single thread will enter each section clause

Single processor region/1



This construct is ideally suited for I/O or initialization

```
for (i=0; i < N; i++)
{
                          Serial
      .....
   "read a[0..N-1]";
      .....
}
```

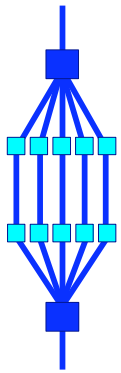
"declare A to be be shared"

```
#pragma omp parallel for
for (i=0; i < N; i++)
{
      .....
        one volunteer requested
   "read a[0..N-1]";
                    thanks, we're done
      .....
}
```

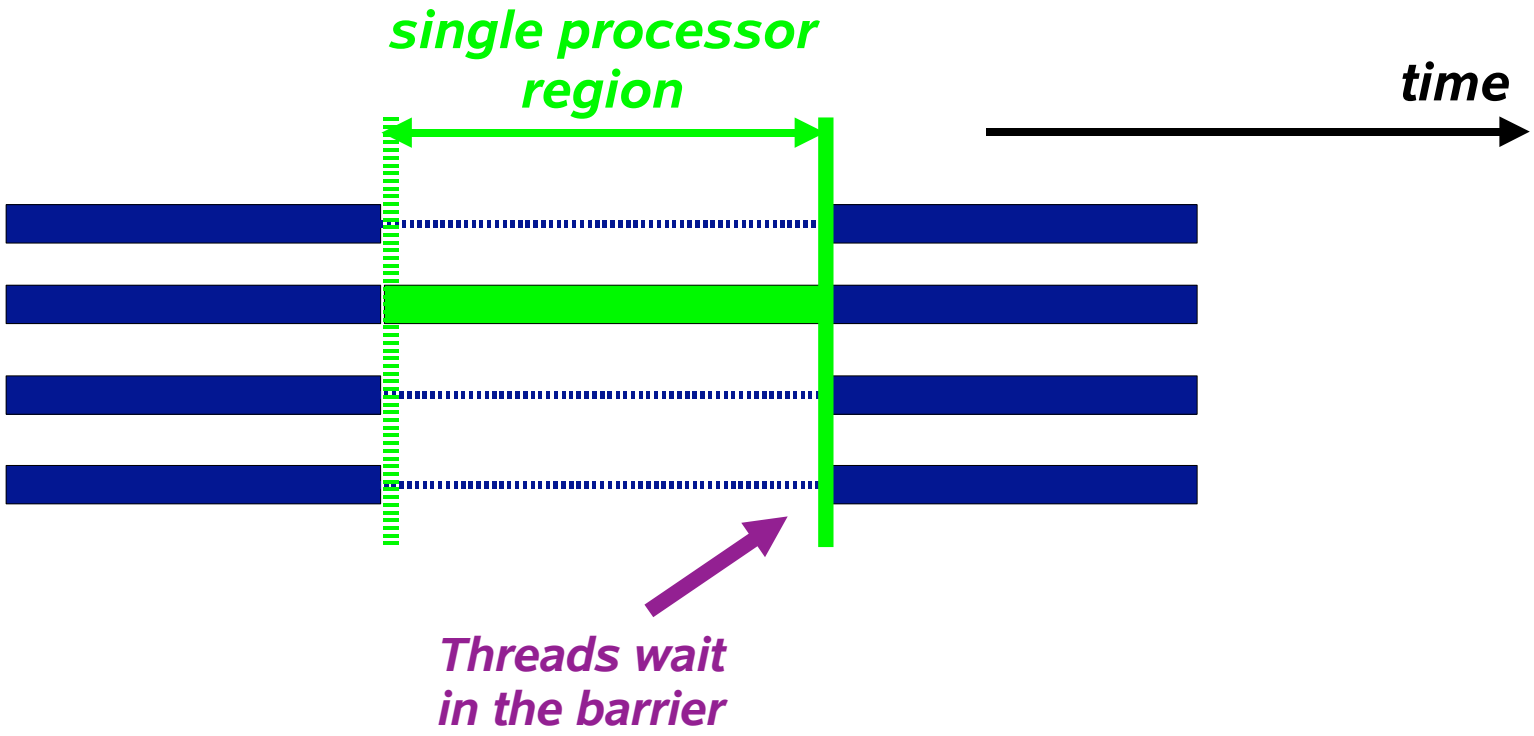
Parallel

May have to insert a barrier here

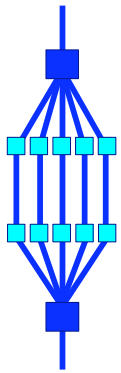
Single processor region/2



- *Usually, there is a barrier needed after this region*
- *Might therefore be a scalability bottleneck (Amdahl's law)*



SINGLE and MASTER construct



Only one thread in the team executes the code enclosed

```
#pragma omp single [clause[[,] clause] ...]
{
        <code-block>
}
```

```
!$omp single [clause[[,] clause] ...]
   <code-block>
!$omp end single [nowait]
```

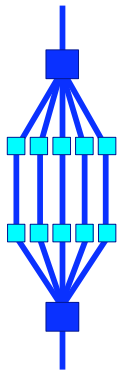
Only the master thread executes the code block:

```
#pragma omp master
{<code-block>}
```

```
!$omp master
        <code-block>
!$omp end master
```

There is no implied barrier on entry or exit !

More synchronization directives



The enclosed block of code is executed in the order in which iterations would be executed sequentially:

```
#pragma omp ordered
{<code-block>}
```

```
!$omp ordered
        <code-block>
!$omp end ordered
```

Expensive !

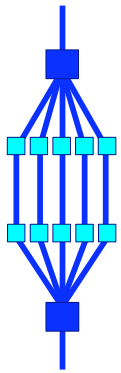
Ensure that all threads in a team have a consistent view of certain objects in memory:

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

In the absence of a list, all visible variables are flushed

OpenMP environment variables



OpenMP environment variable	Default for Sun OpenMP
OMP_NUM_THREADS n	1
OMP_SCHEDULE "schedule,[chunk]"	static, "N/P" (1)
OMP_DYNAMIC { TRUE | FALSE }	TRUE (2)
OMP_NESTED { TRUE | FALSE }	FALSE (3)

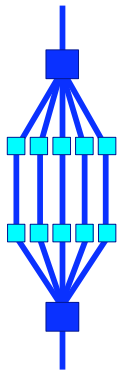
(1) The chunk size approximately equals the number of iterations (N) divided by the number of threads (P)

(2) The number of threads will be limited to the number of on-line processors in the system. This can be changed by setting OMP_DYNAMIC to FALSE.

(3) Multi-threaded execution of inner parallel regions in nested parallel regions is supported as of Sun Studio 10

Note: The names are in uppercase, the values are case insensitive

Global data - example



```
          ....
      include "global.h"
          ....
!$omp parallel private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end do
!$omp end parallel

          ....
```

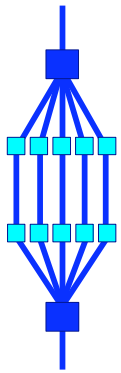
file global.h

```
common /work/a(m,n),b(m)
```

```
subroutine suba(j)
   .....
include "global.h"
   .....
do i = 1, m
   b(i) = j
end do

do i = 1, m
    a(i,j) = func_call(b(i))
end do

return
end
```

Race condition !

Global data - race condition



Thread 1

```
call suba(1)
```

```
subroutine suba(j=1)
do i = 1, m
   b(i) = 1
end do
          ....
do i = 1, m
 a(i,1)=func_call(b(i))
end do
```

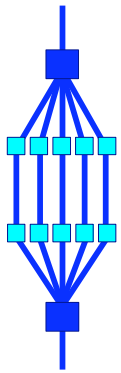
Thread 2

```
call suba(2)
```

```
subroutine suba(j=2)
do i = 1, m
   b(i) = 2
end do
          ....
do i = 1, m
  a(i,2)=func_call(b(i))
end do
```

Shared

Example - solution



```
          ....
      include "global.h"
          ....
!$omp parallel private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end do
!$omp end parallel

          ....
```

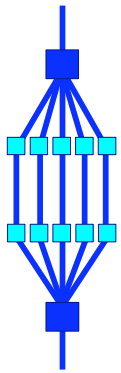
new file global.h

```
integer, parameter:: nthreads=4
common /work/a(m,n)
common /tprivate/b(m,nthreads)
```

```
subroutine suba(j)
  .....
include "global.h"
  .....

TID = omp_get_thread_num()+1
do i = 1, m
   b(i,TID) = j
end do

do i = 1, m
   a(i,j)=func_call(b(i,TID))
end do

return
end
```

- *By expanding array B, we can give each thread unique access to it's storage area*
- *Note that this can also be done using dynamic memory (allocatable, malloc, ...)*

Example - solution 2



```
          ....
      include "global.h"
          ....
!$omp parallel private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end do
!$omp end parallel

          ....
```

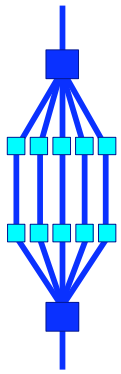
new file global.h

```
common /work/a(m,n)
common /tprivate/b(m)
!$omp threadprivate(/tprivate/)
```

```
subroutine suba(j)
  .....
include "global.h"
  .....

do i = 1, m
   b(i) = j
end do

do i = 1, m
   a(i,j) = func_call(b(i))
end do

return
end
```

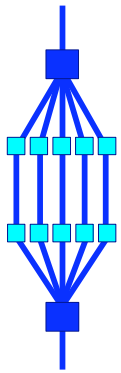
- *The compiler will create thread private copies of array B, to give each thread unique access to it's storage area*
- *Note that the number of copies will be automatically adapted to the number of threads*

About global data



- *Global data is shared and requires special care*
- *A problem may arise in case multiple threads access the same memory section simultaneously:*
  - *Read-only data is no problem*
  - *Updates have to be checked for race conditions*
- *It is your responsibility to deal with this situation*
- *In general one can do the following:*
  - *Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel*
  - *Manually create thread private copies of the latter*
  - *Use the thread ID to access these private copies*
- *Alternative: Use OpenMP's threadprivate construct*

The threadprivate construct



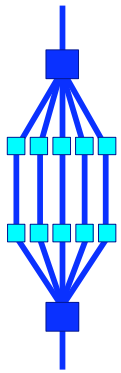
- *OpenMP's threadprivate directive*

```
!$omp threadprivate (/cb/ [,/cb/] ...)
```

```
#pragma omp threadprivate (list)
```

- *Thread private copies of the designated global variables and common blocks will be made*
- *Several restrictions and rules apply when doing this:*
  - *The number of threads has to remain the same for all the parallel regions (i.e. no dynamic threads)*
    - ✔ *Sun implementation supports changing the number of threads*
  - *Initial data is undefined, unless* **copyin** *is used*
  - *.....*
- *Check the documentation when using threadprivate !*

The copyin clause



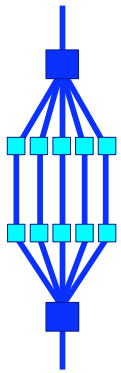
copyin (list)

- ✔ *Applies to THREADPRIVATE common blocks only*
- ✔ *At the start of the parallel region, data of the master thread is copied to the thread private copies*

Example:

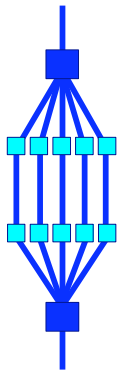
```
      common /cblock/velocity
      common /fields/xfield, yfield, zfield

! create thread private common blocks

!$omp threadprivate (/cblock/, /fields/)

!$omp parallel           &
!$omp default (private)  &
!$omp copyin ( /cblock/, zfield )
```

Runtime library overview



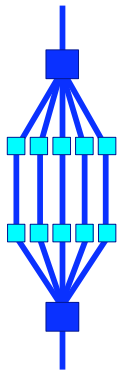
Name	Functionality
omp_set_num_threads	Set number of threads
omp_get_num_threads	Return number of threads in team
omp_get_max_threads	Return maximum number of threads
omp_get_thread_num	Get thread ID
omp_get_num_procs	Return maximum number of processors
omp_in_parallel	Check whether in parallel region
omp_set_dynamic	Activate dynamic thread adjustment
	(but implementation is free to ignore this)
omp_get_dynamic	Check for dynamic thread adjustment
omp_set_nested	Activate nested parallelism
	(but implementation is free ignore this)
omp_get_nested	Check for nested parallelism
omp_get_wtime	Returns wall clock time
omp_get_wtick	Number of seconds between clock ticks

OpenMP locking routines



- *Locks provide greater flexibility over critical sections and atomic updates:*
  - *Possible to implement asynchronous behaviour*
  - *Not block structured*
- *The so-called lock variable, is a special variable:*
  - *Fortran: type INTEGER and of a KIND large enough to hold an address*
  - *C/C++: type omp_lock_t and omp_nest_lock_t for nested locks*
- *Lock variables should be manipulated through the API only*
- *It is illegal, and behaviour is undefined, in case a lock variable is used without the appropriate initialization*

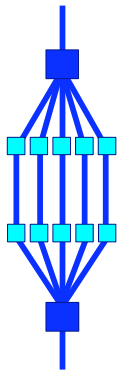
Nested locking

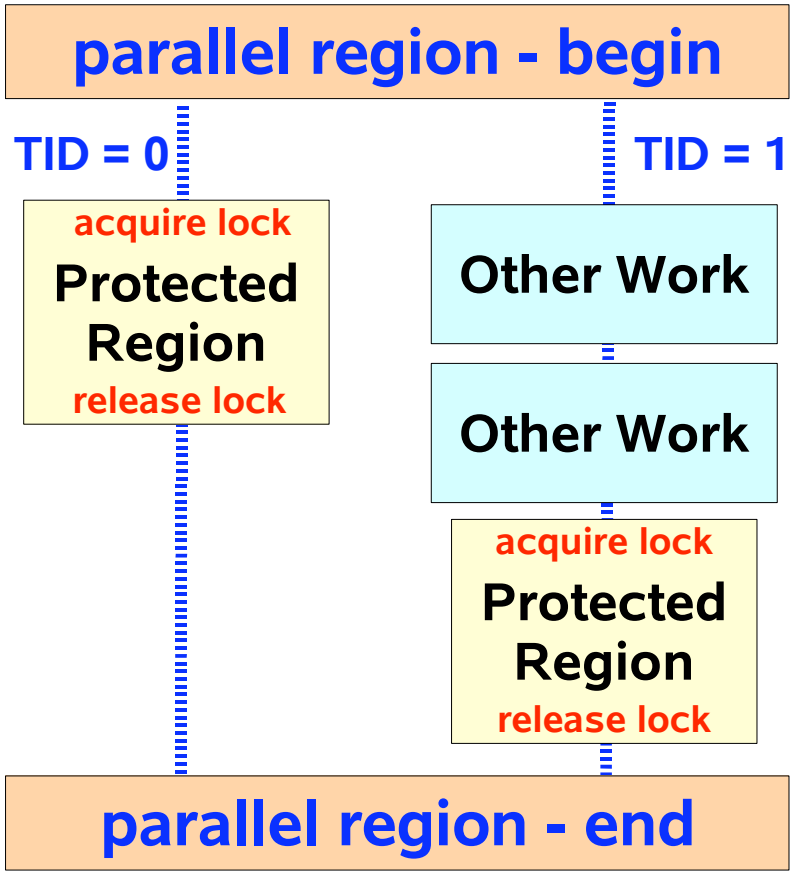


- *Simple locks: may not be locked if already in a locked state*
- *Nestable locks: may be locked multiple times by the same thread before being unlocked*
- *In the remainder, we will discuss simple locks only*
- *The interface for functions dealing with nested locks is similar (but using nestable lock variables):*

Simple locks	Nestable locks
omp_init_lock	omp_init_nest_lock
omp_destroy_lock	omp_destroy_nest_lock
omp_set_lock	omp_set_nest_lock
omp_unset_lock	omp_unset_nest_lock
omp_test_lock	omp_test_nest_lock

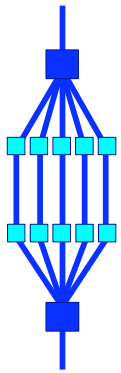
OpenMP locking example





- *The protected region contains the update of a shared variable*
- *One thread will acquire the lock and perform the update*
- *Meanwhile, the other thread will do some other work*
- *When the lock is released again, the other thread will perform the update*

Locking example - the code



```
      Program Locks
            ....
      Call omp_init_lock (LCK)

!$omp parallel shared(SUM,LCK) private(TID)

       TID = omp_get_thread_num()

       Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
          Call Do_Something_Else(TID)
       End Do

       Call Do_Work(SUM,TID)

       Call omp_unset_lock (LCK)

!$omp end parallel

      Call omp_destroy_lock (LCK)

      Stop
      End
```

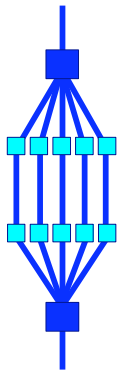
Initialize lock variable

Check availability of lock (will also set the lock)

Release lock again

Remove lock association

Example output for 2 threads



```
 TID:  1 at 09:07:27 => entered parallel region
 TID:  1 at 09:07:27 => done with WAIT loop and has the lock
 TID:  1 at 09:07:27 => ready to do the parallel work
 TID:  1 at 09:07:27 => this will take about 18 seconds
 TID:  0 at 09:07:27 => entered parallel region
 TID:  0 at 09:07:27 =>   WAIT for lock - will do something else for  5 seconds
 TID:  0 at 09:07:32 =>   WAIT for lock - will do something else for  5 seconds
 TID:  0 at 09:07:37 =>   WAIT for lock - will do something else for  5 seconds
 TID:  0 at 09:07:42 =>   WAIT for lock - will do something else for  5 seconds
 TID:  1 at 09:07:45 => done with my work
 TID:  1 at 09:07:45 => done with work loop - released the lock
 TID:  1 at 09:07:45 => ready to leave the parallel region
 TID:  0 at 09:07:47 => done with WAIT loop and has the lock
 TID:  0 at 09:07:47 => ready to do the parallel work
 TID:  0 at 09:07:47 => this will take about 18 seconds
 TID:  0 at 09:08:05 => done with my work
 TID:  0 at 09:08:05 => done with work loop - released the lock
 TID:  0 at 09:08:05 => ready to leave the parallel region
Done at 09:08:05 - value of SUM is 1100
```

Used to check the answer

Note: program has been instrumented to get this information

Bug examples

```
int i, chunk, tid;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;;

#pragma omp parallel for     \
  shared(a,b,c,chunk)          \
  private(i,tid)
  {
  tid = omp_get_thread_num();
  for (i=0; i < N; i++)
    {
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
    }
  }  /* end of parallel for construct */

}
```

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

Bug examples

```
int i, chunk, tid;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;;

#pragma omp parallel for     \
  shared(a,b,c,chunk)          \
  private(i,tid)
  {
  tid = omp_get_thread_num();
  for (i=0; i < N; i++)
    {
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
    }
  }  /* end of parallel for construct */

}
```

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

Bug examples

```
int i, chunk, tid;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;;

#pragma omp parallel for     \
  shared(a,b,c,chunk)          \
  private(i,tid)
  {
  tid = omp_get_thread_num();
  for (i=0; i < N; i++)
    {
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
    }
  }  /* end of parallel for construct */

}
```

```
int i, chunk, tid;
float a[N], b[N], c[N];
char first_time;

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
first_time = 'y';

#pragma omp parallel for     \
  shared(a,b,c,chunk)        \
  private(i,tid)             \
  firstprivate(first_time)

  for (i=0; i < N; i++)
    {
    if (first_time == 'y')
      {
      tid = omp_get_thread_num();
      first_time = 'n';
      }
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
    }

}
```

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

Bug examples

```
int nthreads, i, tid;
float total;

/*** Spawn parallel region ***/
#pragma omp parallel
  {
  /* Obtain thread number */
  tid = omp_get_thread_num();
  /* Only master thread does this */
  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d is starting...\n",tid);

  #pragma omp barrier

  /* do some work */
  total = 0.0;
  #pragma omp for
  for (i=0; i<1000000; i++)
     total = total + i*1.0;

  printf ("Thread %d is done! Total= %e\n",tid,total);

  } /*** End of parallel region ***/
}
```

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

Bug examples

```
int nthreads, i, tid;
float total;

/*** Spawn parallel region ***/
#pragma omp parallel
  {
  /* Obtain thread number */
  tid = omp_get_thread_num();
  /* Only master thread does this */
  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d is starting...\n",tid);

  #pragma omp barrier

  /* do some work */
  total = 0.0;
  #pragma omp for
  for (i=0; i<1000000; i++)
     total = total + i*1.0;

  printf ("Thread %d is done! Total= %e\n",tid,total);

  } /*** End of parallel region ***/
}
```

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

Bug examples

```
int nthreads, i, tid;
float total;

/*** Spawn parallel region ***/
#pragma omp parallel
  {
  /* Obtain thread number */
  tid = omp_get_thread_num();
  /* Only master thread does this */
  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d is starting...\n",tid);

  #pragma omp barrier

  /* do some work */
  total = 0.0;
  #pragma omp for
  for (i=0; i<1000000; i++)
     total = total + i*1.0;

  printf ("Thread %d is done! Total= %e\n",tid,total);

  } /*** End of parallel region ***/
}
```

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

```
int nthreads, i, tid;
float total;

/*** Spawn parallel region ***/
#pragma omp parallel
  {
  /* Obtain thread number */
  tid = omp_get_thread_num();
  /* Only master thread does this */
  if (tid == 0) {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d is starting...\n",tid);

  #pragma omp barrier

  /* do some work */
  total = 0.0;
  #pragma omp for reduction(+:total)
  for (i=0; i<1000000; i++)
     total = total + i*1.0;

  printf ("Thread %d is done! Total= %e\n",tid,total);

  } /*** End of parallel region ***/
}
```

Segmentation fault

```
#define N 1048
int main (int argc, char *argv[]) {
int nthreads, tid, i, j;
double a[N][N];

/* Fork a team of threads with explicit variable scoping */
#pragma omp parallel shared(nthreads) private(i,j,tid,a)
  {

  /* Obtain/print thread info */
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n", tid);

  /* Each thread works on its own private copy of the array */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      a[i][j] = tid + i + j;
  /* For confirmation */
  printf("Thread %d done. Last element= %f\n",tid,a[N-1][N-1]);
  }  /* All threads join master thread and disband */
}
```

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

Bug examples

```
#define N 1048
int main (int argc, char *argv[]) {
int nthreads, tid, i, j;
double a[N][N];

/* Fork a team of threads with explicit variable scoping */
#pragma omp parallel shared(nthreads) private(i,j,tid,a)
  {

  /* Obtain/print thread info */
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n", tid);

  /* Each thread works on its own private copy of the array */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      a[i][j] = tid + i + j;
  /* For confirmation */
  printf("Thread %d done. Last element= %f\n",tid,a[N-1][N-1]);
  }  /* All threads join master thread and disband */
}
```

```
#define N 1048
int main (int argc, char *argv[]) {
int nthreads, tid, i, j;
double ***a;

#pragma omp parallel private(nthreads)
nthreads=omp_get_num_threads();
a=(double***) allocate3(n,n,nth);
/* Fork a team of threads with explicit variable scoping */
#pragma omp parallel shared(nthreads,a) private(i,j,tid)
  {

  /* Obtain/print thread info */
  tid = omp_get_thread_num();
  printf("Thread %d starting...\n", tid);

  /* Each thread works on its own private copy of the array */
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      a[tid][i][j] = tid + i + j;
  /* For confirmation */
  printf("Thread %d done. Last element= %f\n",tid,a[N-1][N-1]);
  }  /* All threads join master thread and disband */
}
```

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

Serial code

```
float alpha;
void main(int argc, char **argv){
.
.
.
in=(float**)alloc_2d_float(n1,n2);
work=(float*)alloc_1d_float(n1);
out=(float**) alloc_2d_float(n1,n2);

for(i=0; i < n2; i++){
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[j]...)
  }
}
```

Determine where to parallelize

```
float alpha;
void main(int argc, char **argv){
.
.
.
in=(float**)alloc_2d_float(n1,n2);
work=(float*)alloc_1d_float(n1);
out=(float**) alloc_2d_float(n1,n2);

 for(i=0; i < n2; i++){
   for(j=0; j < n1; j++){
     .
     .
     alpha=func(in[i][j]....)
     work[j]=func(in[i][j].....)
     .
     .
     out[i][j]=func(work[j]...)
   }
}
```

Put serial portion in separate function

```
float alpha;
void main(int argc, char **argv){
.
.
.
in=(float**)alloc_2d_float(n1,n2);
work=(float*)alloc_1d_float(n1);
out=(float**) alloc_2d_float(n1,n2);

for(i=0; i < n2; i++){
  serial_work(in,out,work,i);
}
```

```
void serial_work(float **in, float **out, float *work, int i){
.
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[j]...)
  }
}
```

Get rid of global variables modified in serial portion

```
float alpha;
void main(int argc, char **argv){
.
.
.
in=(float**)alloc_2d_float(n1,n2);
work=(float*)alloc_1d_float(n1);
out=(float**) alloc_2d_float(n1,n2);

for(i=0; i < n2; i++){
  serial_work(in,out,work,i);
}
```

```
void serial_work(float **in, float **out, float *work, int i){
.
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[j]...)
  }
}
```

Get rid of global variables modified in serial portion

```
void main(int argc, char **argv){
.
.
.
in=(float**)alloc_2d_float(n1,n2);
work=(float*)alloc_1d_float(n1);
out=(float**) alloc_2d_float(n1,n2);

for(i=0; i < n2; i++){
  serial_work(in,out,work,i);
}
```

```
void serial_work(float **in, float **out, float *work, int i){
  float alpha;
  .
    for(j=0; j < n1; j++){
      .
      .
      alpha=func(in[i][j]....)
      work[j]=func(in[i][j].....)
      .
      .
      out[i][j]=func(work[j]...)
    }
  }
```

Find intermediate heap arrays

```
void main(int argc, char **argv){
.
.
.
in=(float**)alloc_2d_float(n1,n2);
work=(float*)alloc_1d_float(n1);
out=(float**) alloc_2d_float(n1,n2);

for(i=0; i < n2; i++){
  serial_work(in,out,work,i);
}
```

```
void serial_work(float **in, float **out, float *work, int i){
float alpha;
.
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[j]...)
  }
}
```

Expand dimensionality by nthreads

```
void main(int argc, char **argv){
.
.

  nth=1; //For serial case
  in=(float**)alloc_2d_float(n1,n2);
  work=(float**)alloc_2d_float(n1,nth);
  out=(float**) alloc_2d_float(n1,n2);

  ith=0;
  for(i=0; i < n2; i++){
    serial_work(in,out,work,i,ith);
  }
```

```
void serial_work(float **in, float **out, float **work, int i,
int ith){
float alpha;
.
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[ith][j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[ith][j]...)
  }
}
```

Figure out number of threads

```
void main(int argc, char **argv){
.
.

nth=1; //For serial case
#pragma omp parallel
nth=omp_get_num_threads();
#pragma omp end parallel

in=(float**)alloc_2d_float(n1,n2);
work=(float**)alloc_2d_float(n1,nth);
out=(float**) alloc_2d_float(n1,n2);

ith=0;
for(i=0; i < n2; i++){
  serial_work(in,out,work,i,ith);
}
```

```
void serial_work(float **in, float **out, float **work, int i,
int ith){
float alpha;
.
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[ith][j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[ith][j]...)
  }
}
```

Parallelize loop

```
void main(int argc, char **argv){
.
.

nth=1; //For serial case
#pragma omp parallel
nth=omp_get_num_threads();
#pragma omp end parallel

in=(float**)alloc_2d_float(n1,n2);
work=(float**)alloc_2d_float(n1,nth);
out=(float**) alloc_2d_float(n1,n2);

ith=0;
#pragma omp parallel for private(i)
for(i=0; i < n2; i++){
  serial_work(in,out,work,i,ith);
}
```

```
void serial_work(float **in, float **out, float **work, int i,
int ith){
float alpha;
.
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[ith][j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[ith][j]...)
  }
}
```

Grab thread number

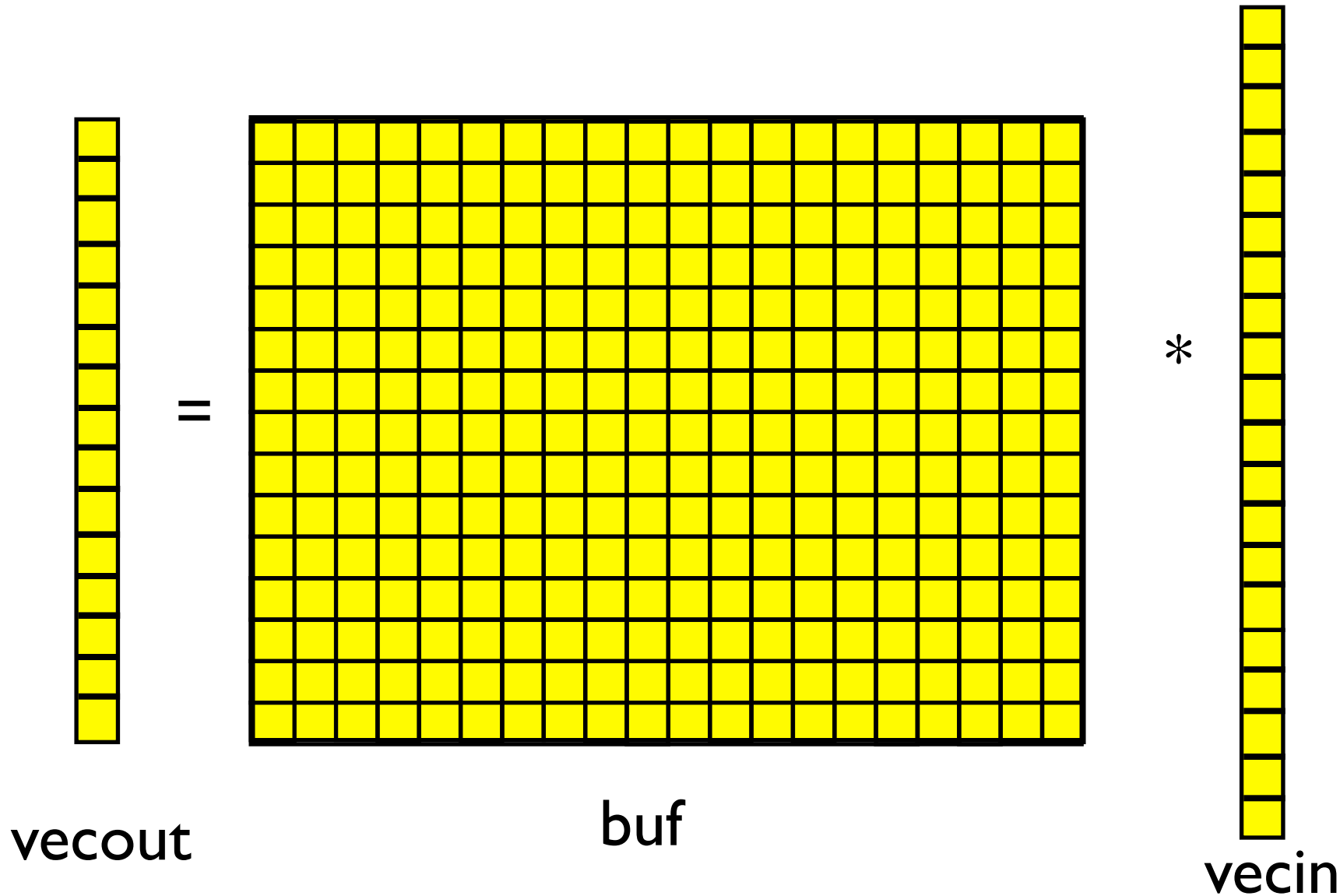
```
void main(int argc, char **argv){
.
.

nth=1; //For serial case
#pragma omp parallel
nth=omp_get_num_threads();
#pragma omp end parallel

in=(float**)alloc_2d_float(n1,n2);
work=(float**)alloc_2d_float(n1,nth);
out=(float**) alloc_2d_float(n1,n2);

#pragma omp parallel for private(i)
for(i=0; i < n2; i++){
  ith=omp_get_thread_num();
  serial_work(in,out,work,i,ith);
}
```

```
void serial_work(float **in, float **out, float **work, int i,
int ith){
float alpha;
.
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[ith][j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[ith][j]...)
  }
}
```

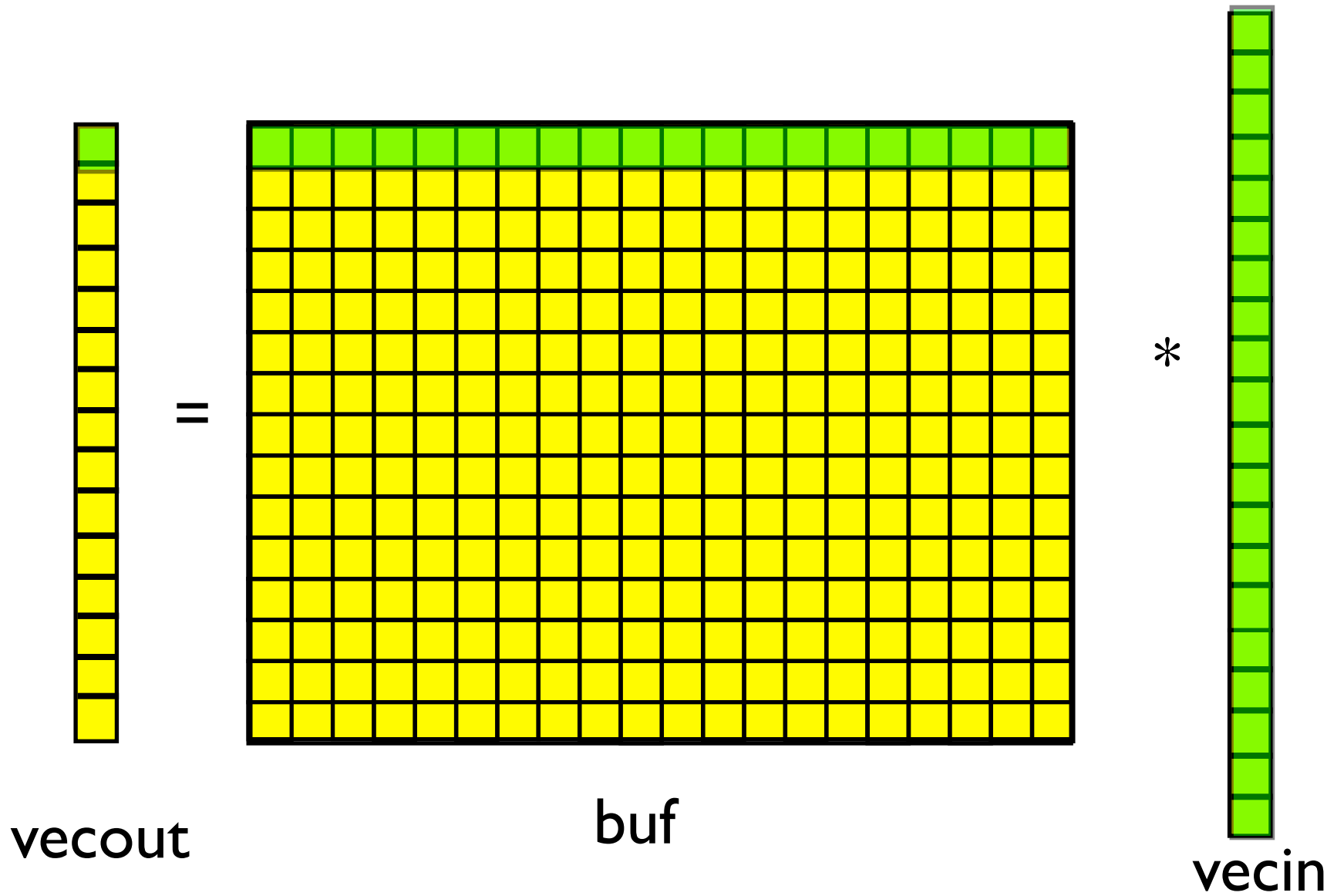
More complex examples

- Matrix-vector multiply
- Sparse matrix vector multiply
- Matrix-matrix multiply
- Convolution

Matrix-vector multiply



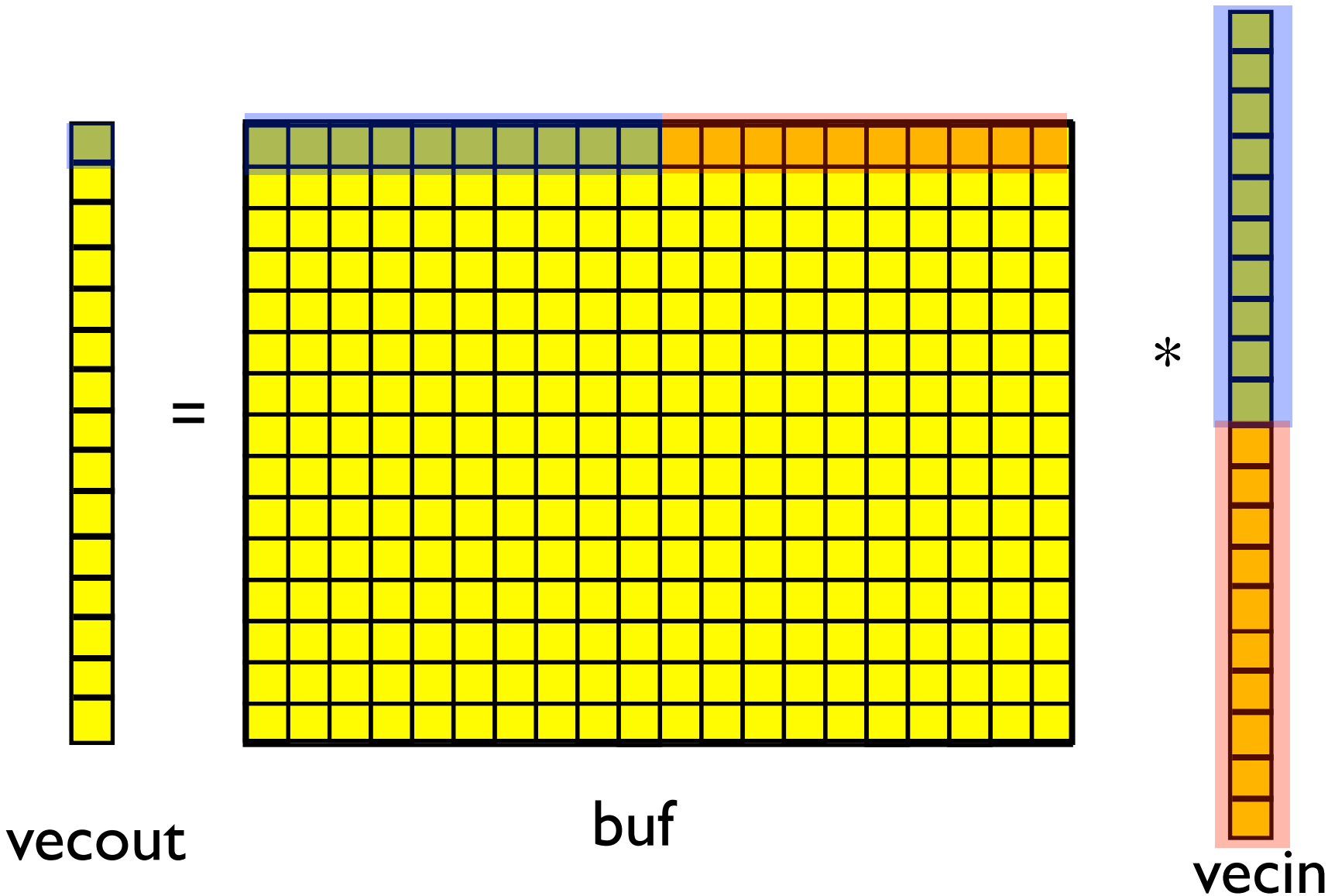
Matrix-vector multiply



Serial code

```
do i2=1,n2
  do i1=1,n1
    rsum=rsum+buf(i1,i2)*vecin(i1)
  end do
  vecout(i2)=rsum
end do
```

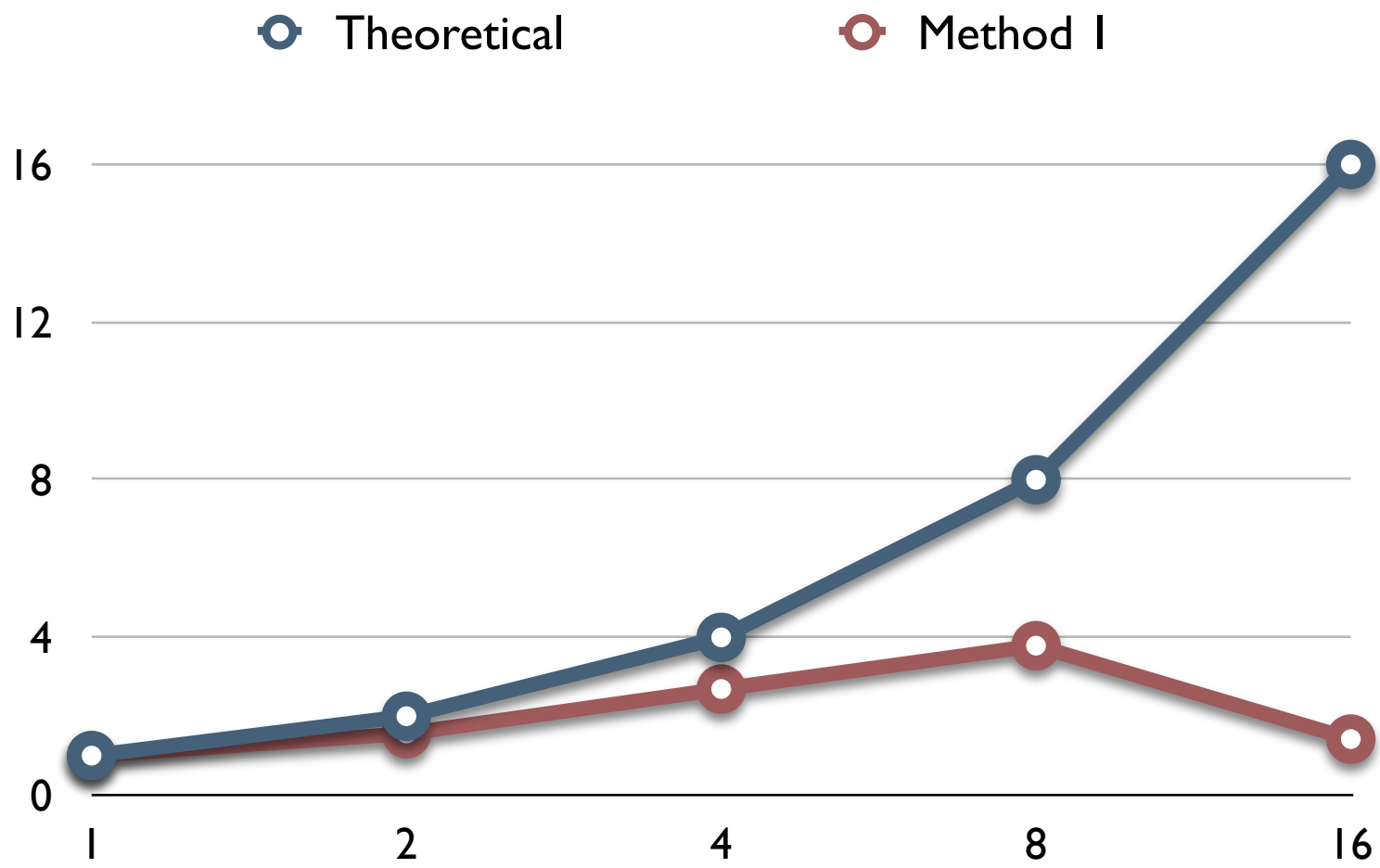
Parallelize over inner axis



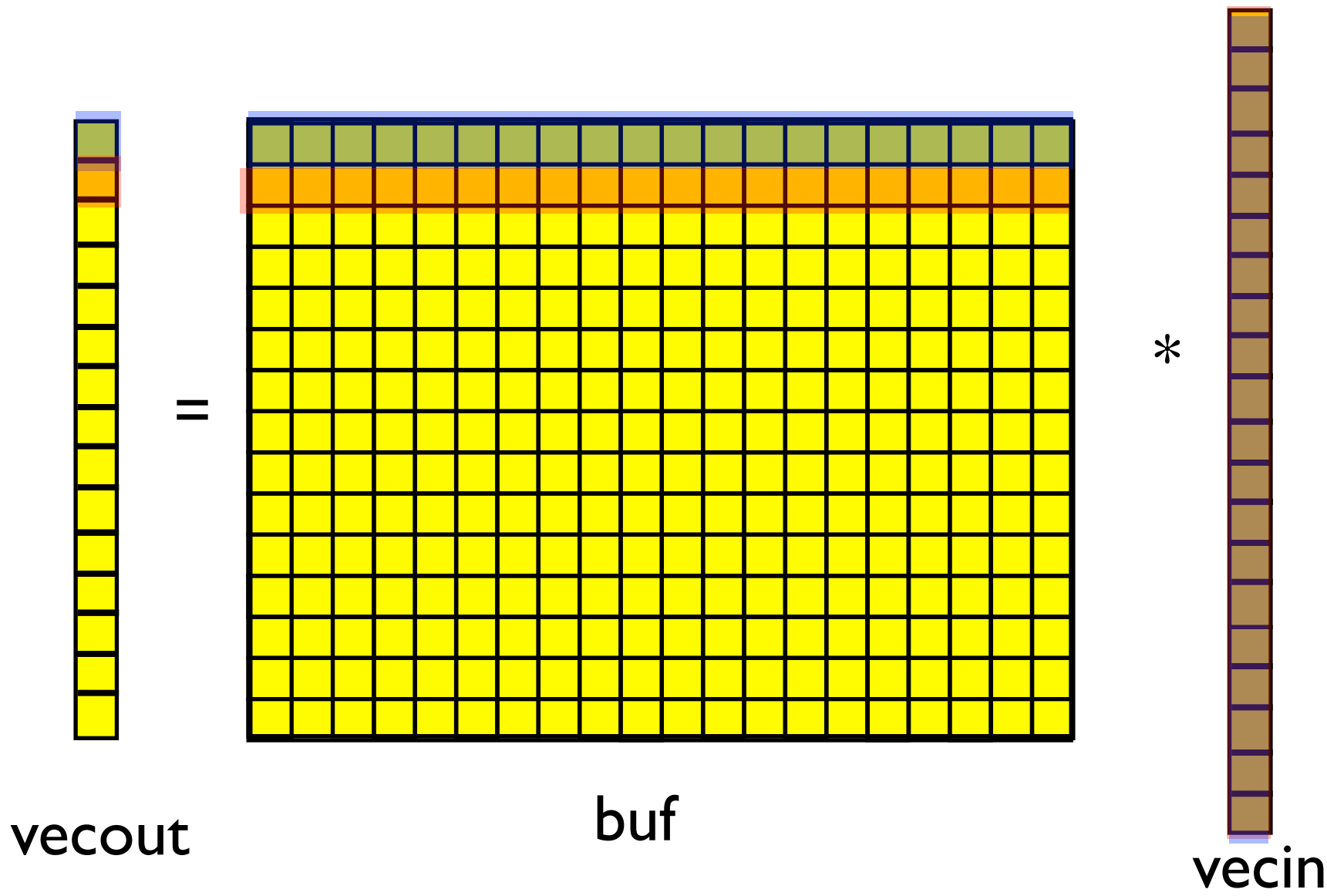
Parallelize over inner access

```
do i2=1,n2
  !$OMP PARALLEL DO private(i1) reduction (+:rsum)
  do i1=1,n1
    rsum=rsum+buf(i1,i2)*vecin(i1)
  end do
  !OMP END PARALLEL DO
  vecout(i2)=rsum
end do
```

Speedup



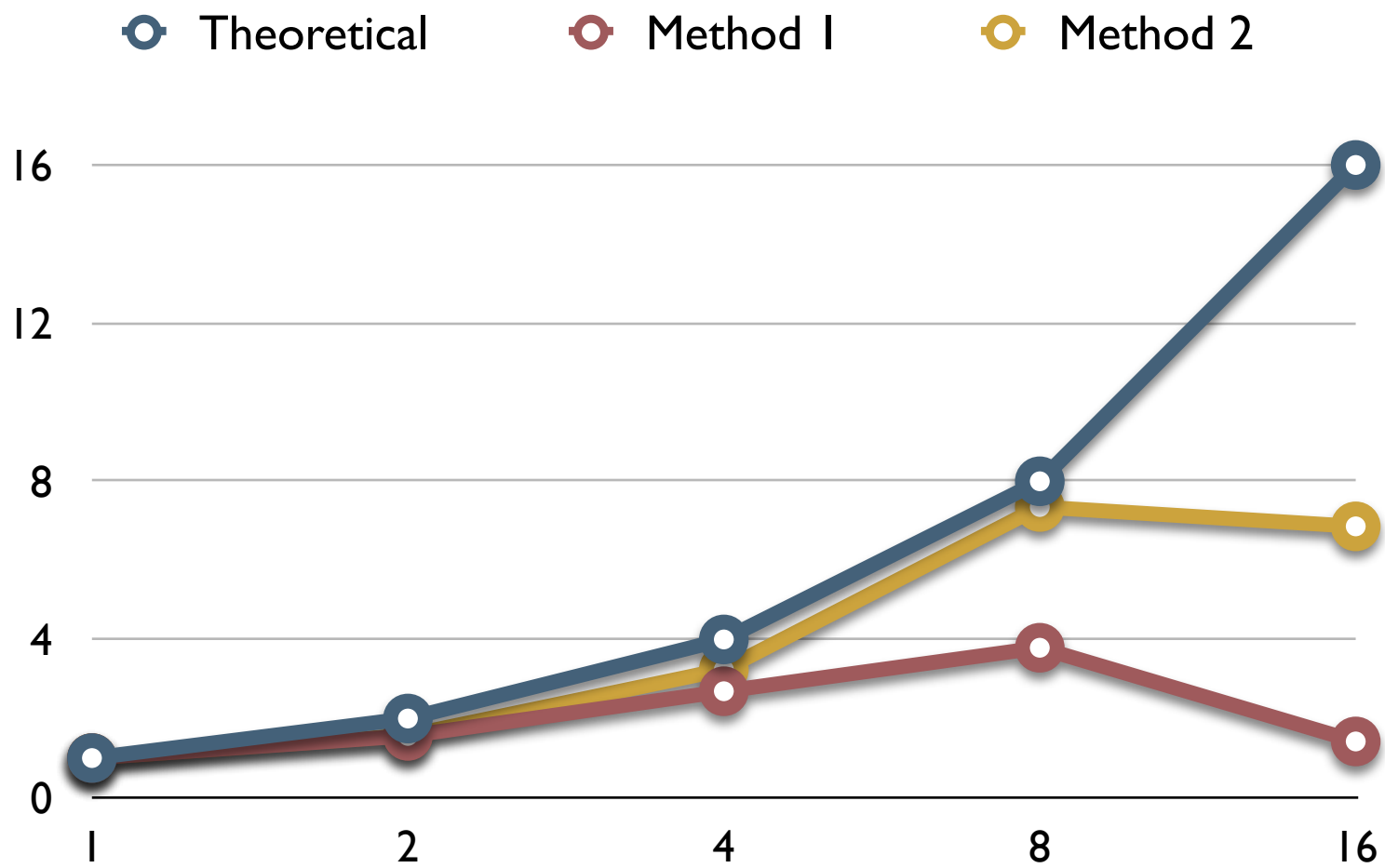
Parallelize over outer axis



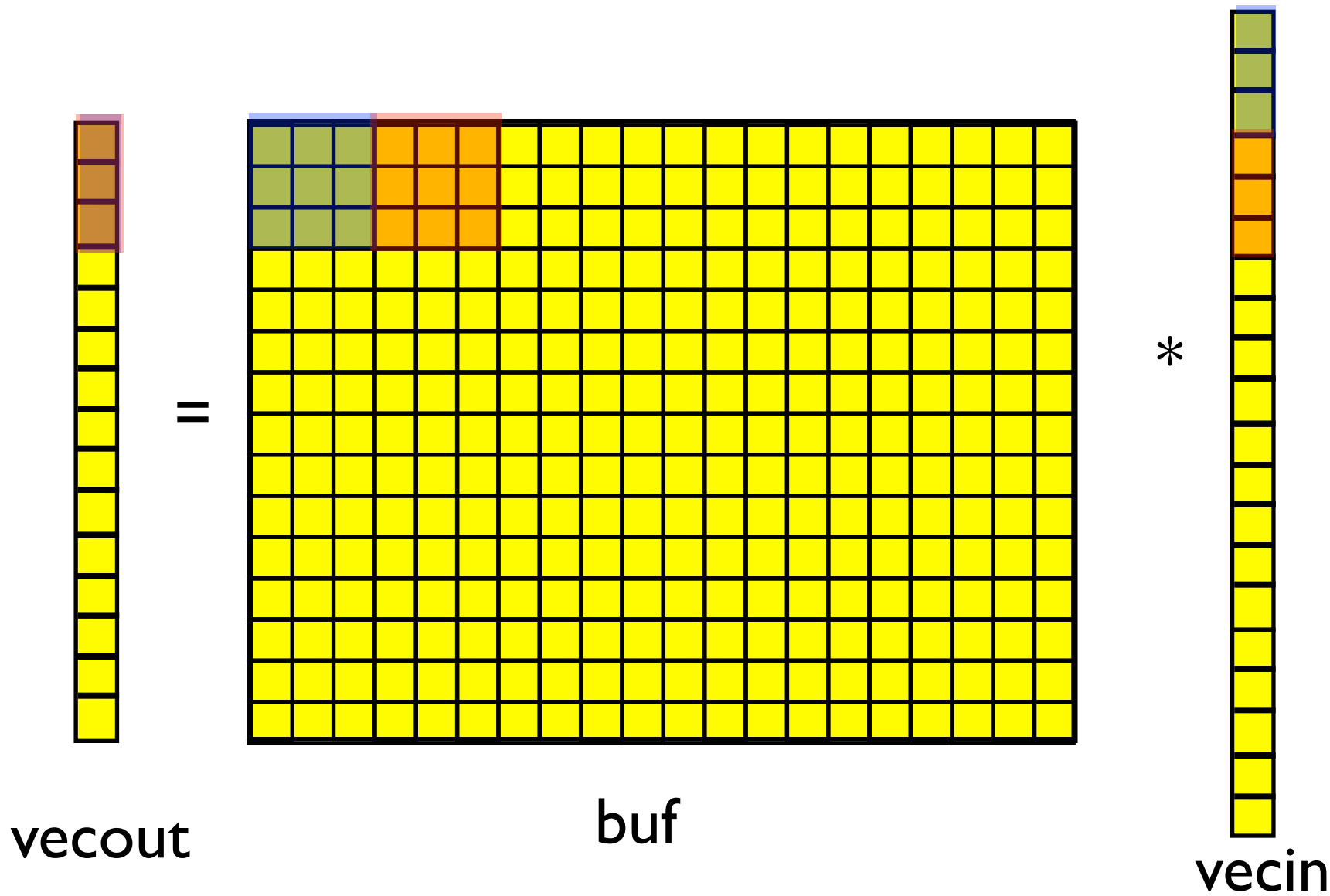
Parallelize over inner access

```
!$OMP PARALLEL DO private(rsum,i2,i1)
 do i2=1,n2
    rsum=0
    do i1=1,n1
      rsum=rsum+buf(i1,i2)*vecin(i1)
    end do
    vecout(i2)=rsum
  end do
 !OMP END PARALLEL DO
```

Speedup



Parallelize over outer axis

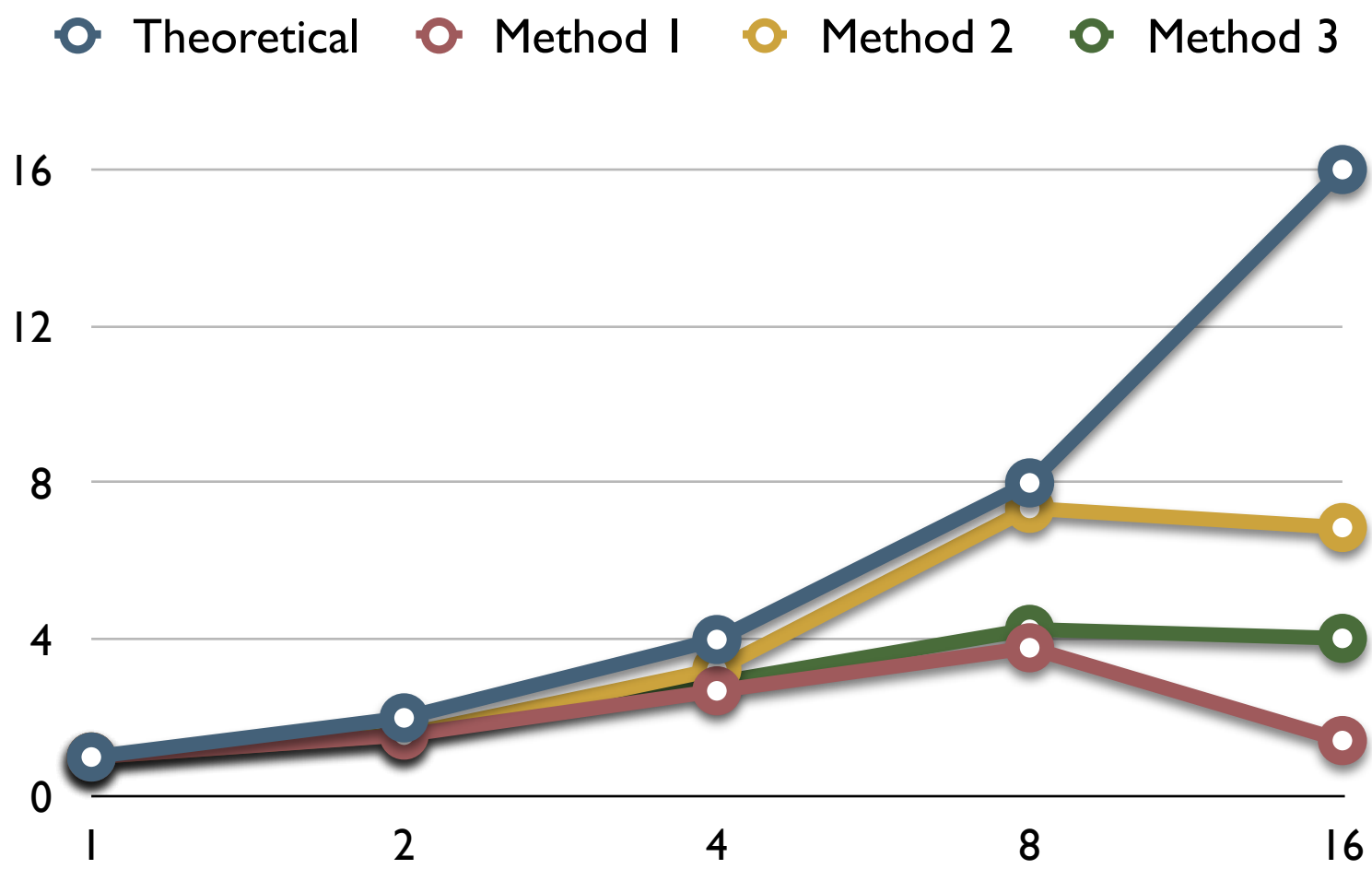


Blocking

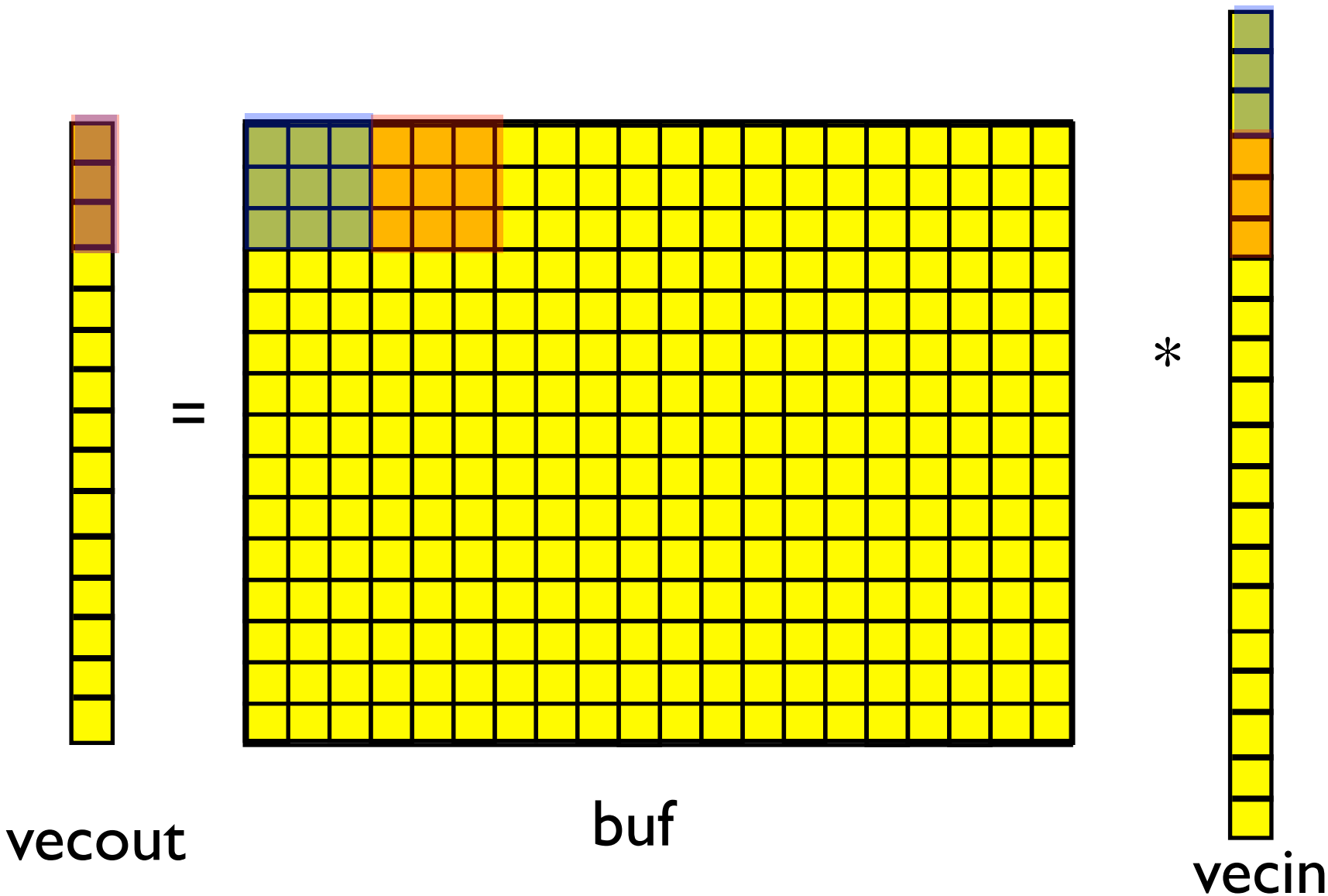
```
i=0
do i2=1,nbig2
 do i1=1,nbig1
   i=i+1
   ibeg(:,i)=(/nbuf1*(i1-1),nbuf2*(i2-1)/)
 end do
end do

!$OMP PARALLEL DO  private(i1) schedule(dynamic)
  do i=1,nbig1*nbig2
      call  sub_mult(buf,vecin,vecout,nbuf1,nbuf2,ibeg(1,i),ibeg(2,i))
    end do
!OMP END PARALLEL DO
```

```
subroutine sub_mult(buf,vecin,vecout,n1,n2,i1beg,i2beg)
  real :: buf(:,:),vecin(:),vecout(:)
  integer :: n1,n2,i1beg,i2beg
  integer :: i1,i2
  real  :: rsum


  do i2=1,n2
    rsum=0
    do i1=1,n1
      rsum=rsum+buf(i1+i1beg,i2+i2beg)
    end do
    !OMP ATOMIC
    vecout(i2)=vecout(i2)+rsum
  end do
end subroutine
```

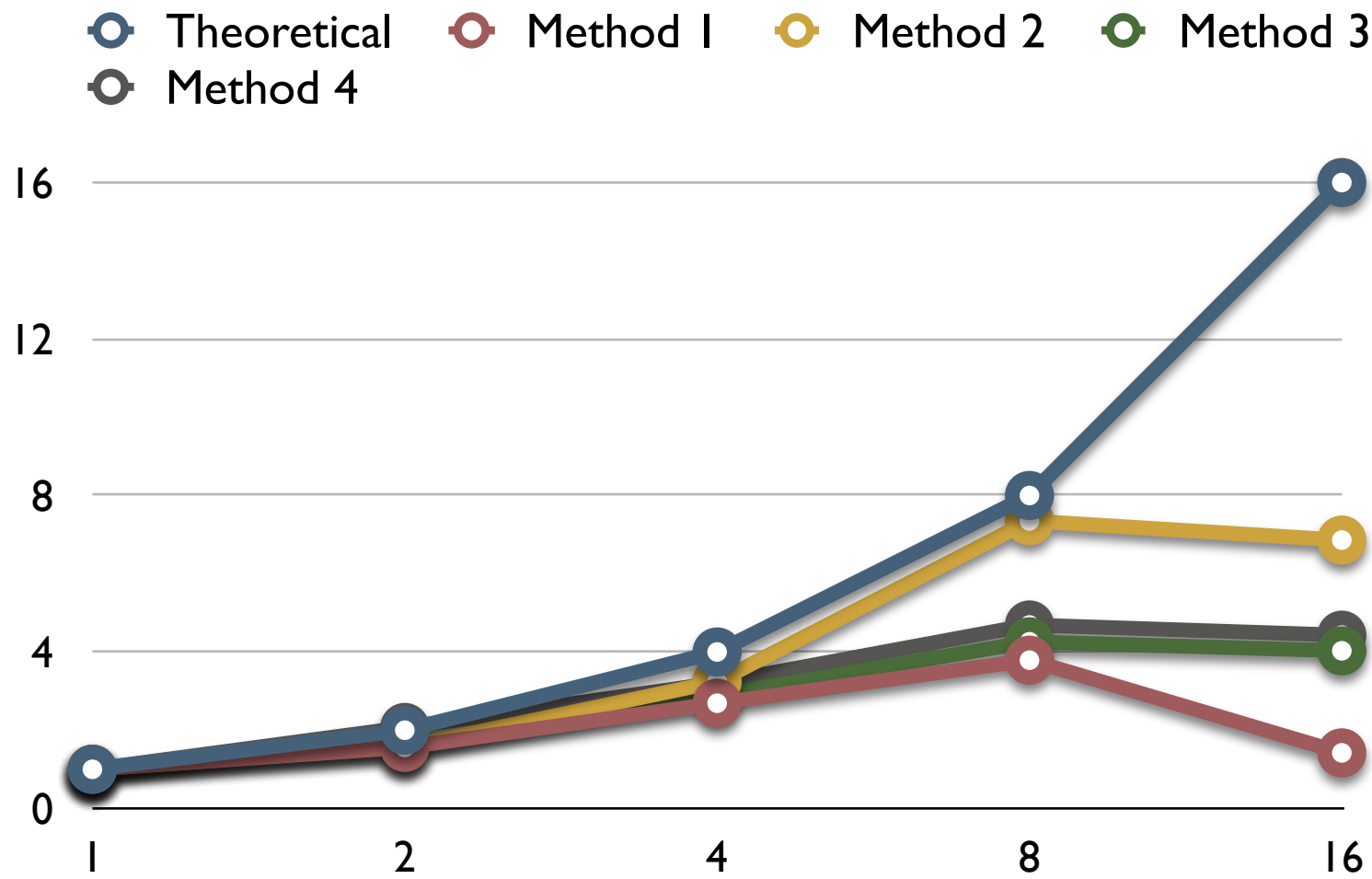
Speedup



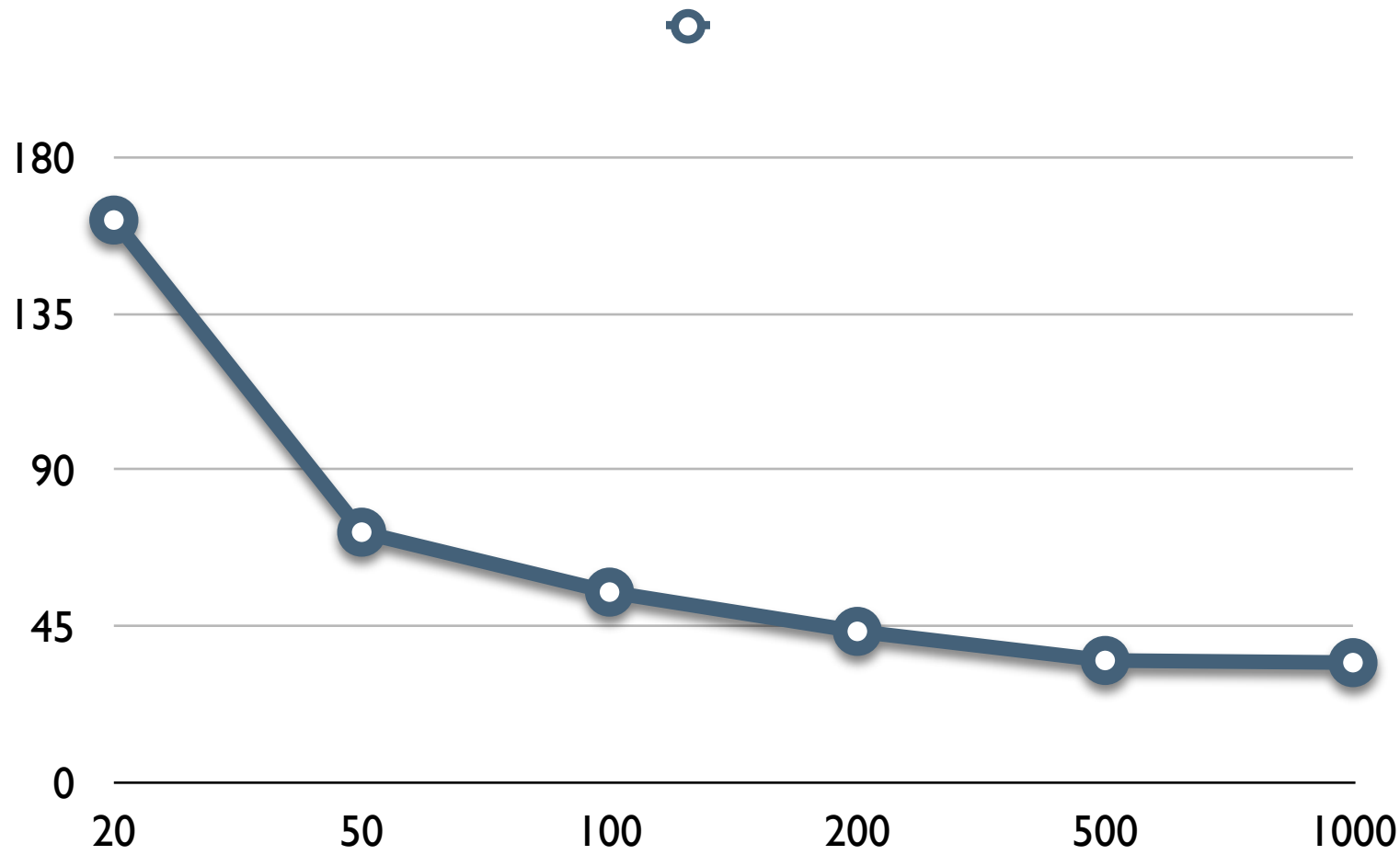
Dynamic blocking



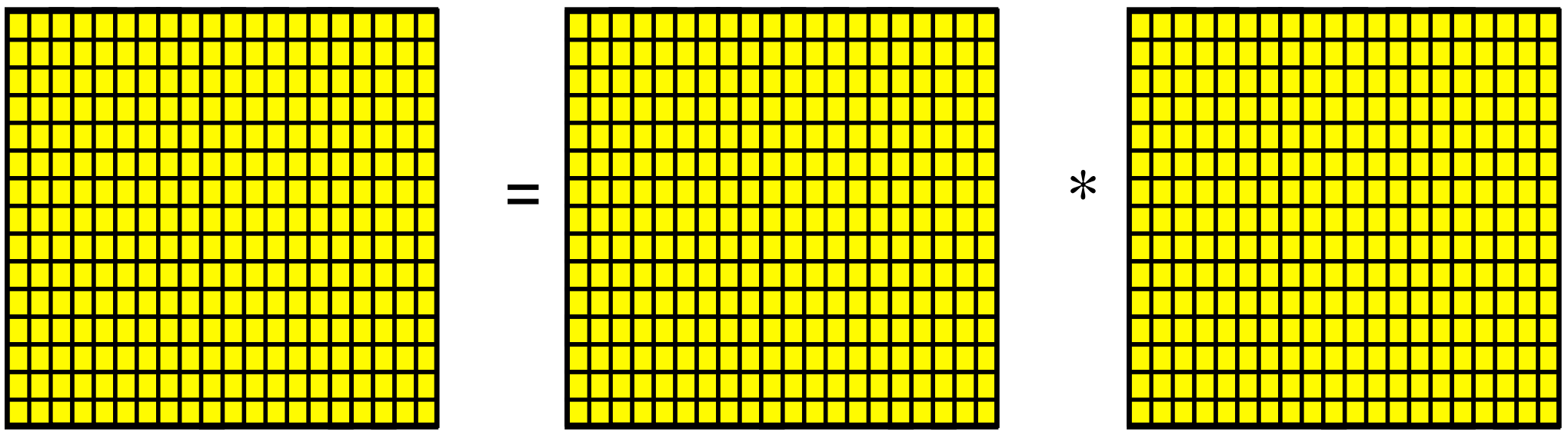
Speedup


Theoretical
Method 1
Method 2
Method 3
Method 4
16
12
8
4
0
1
2
4
8
16

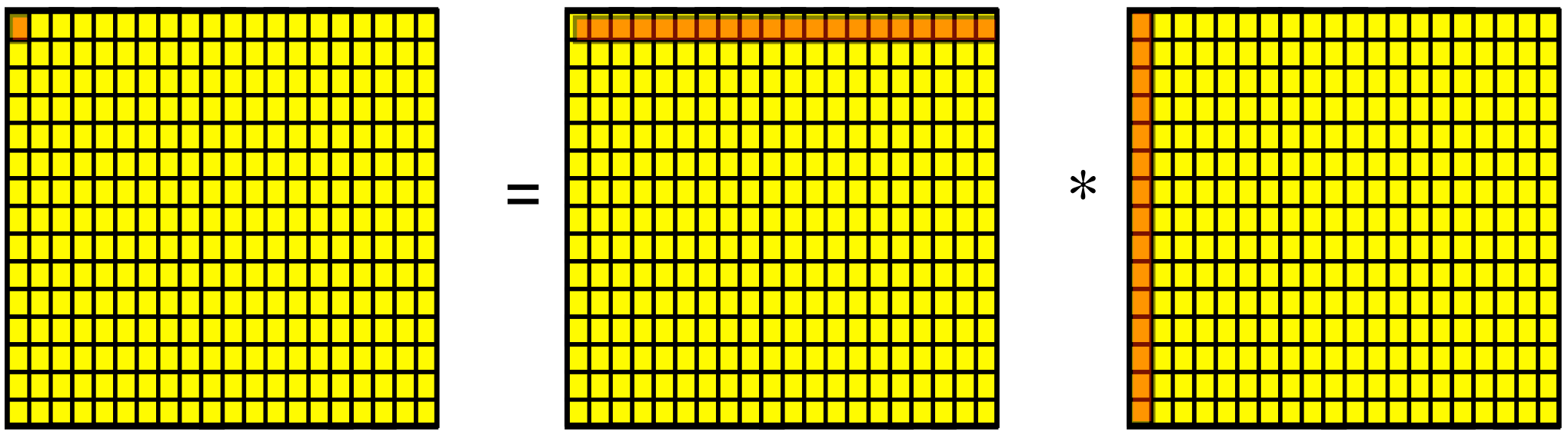
Buffer size



Matrix-vector multiply



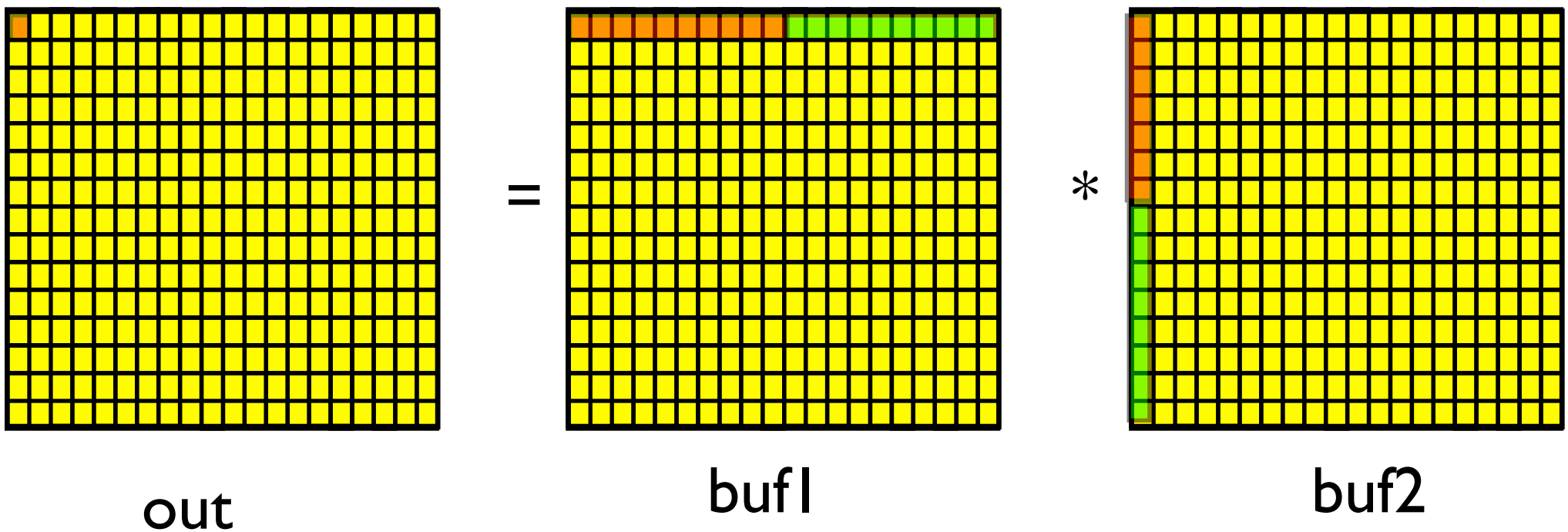
Matrix-matrix multiply



Matrix multiplication

```
do i3=1,n
  do i2=1,n
    rsum=0
    do i1=1,n
      rsum=rsum+in1(i1,i3)*in2(i2,i1)
    end do
    out(i2,i3)=rsum
  end do
end do
```

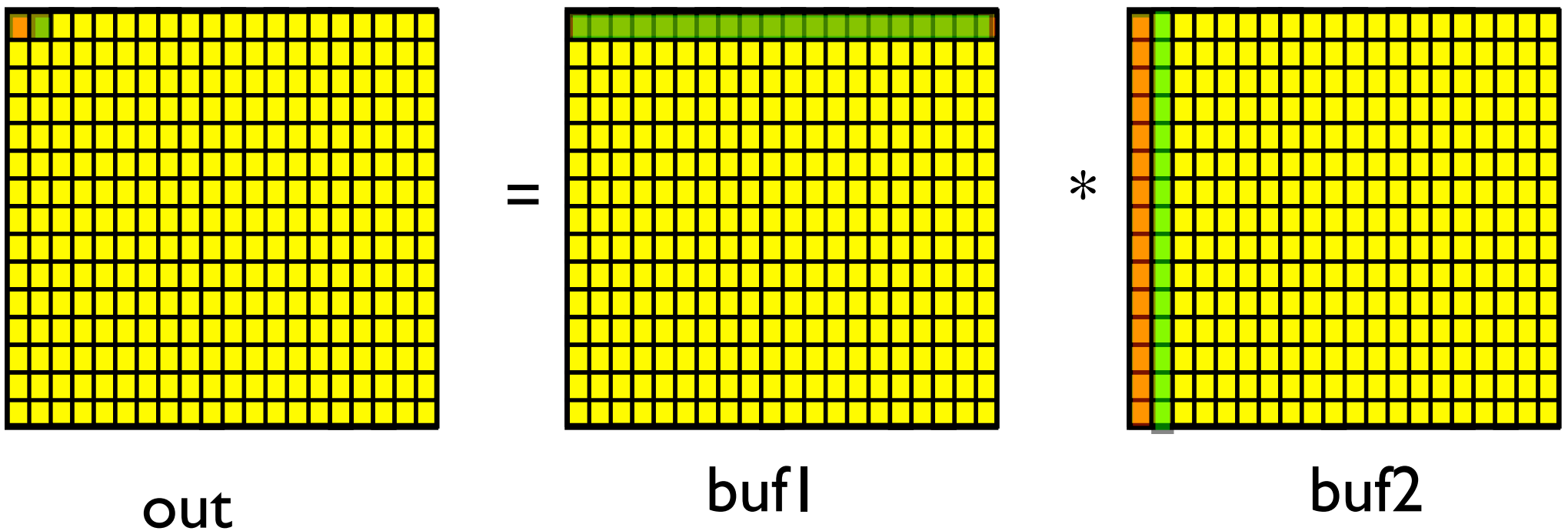
Matrix-vector multiply



Matrix multiplication

```
do i3=1,n
  do i2=1,n
    rsum=0
    !OMP PARALLEL DO private(i1) reduction(+:rsum)
    do i1=1,n
      rsum=rsum+in1(i1,i3)*in2(i2,i1)
    end do
    !OMP END PARALLEL DO
    out(i2,i3)=rsum
  end do
end do
```

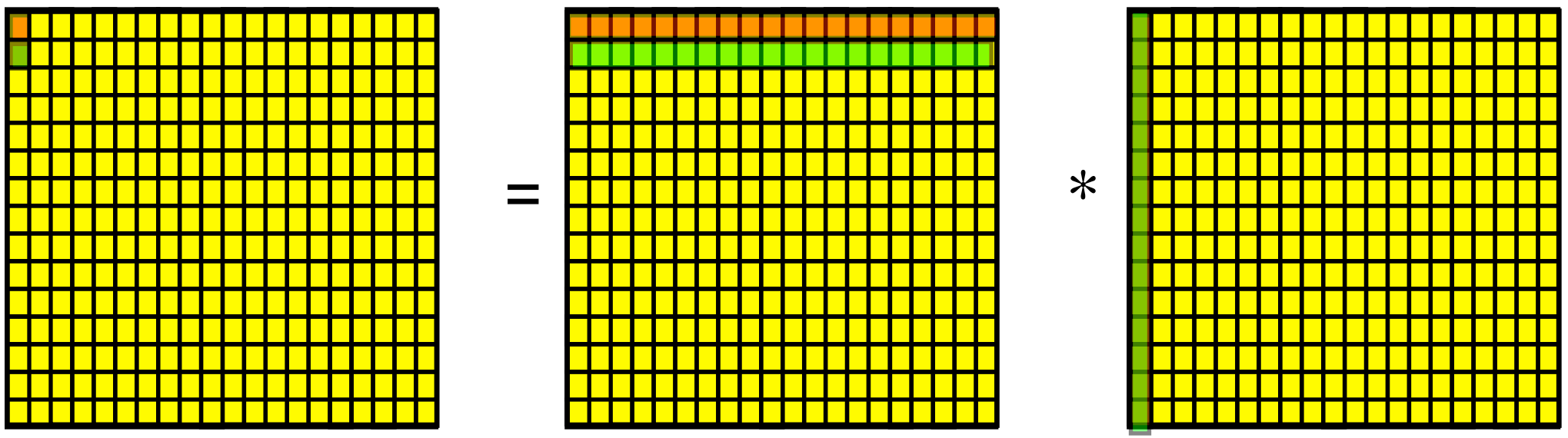
Matrix-vector multiply



Matrix multiplication

```
do i3=1,n
  !OMP PARALLEL DO private(rsum,i2,i1)
  do i2=1,n
     rsum=0
     do i1=1,n
       rsum=rsum+in1(i1,i3)*in2(i2,i1)
     end do
     out(i2,i3)=rsum
   end do
   !OMP END PARALLEL DO
 end do
```

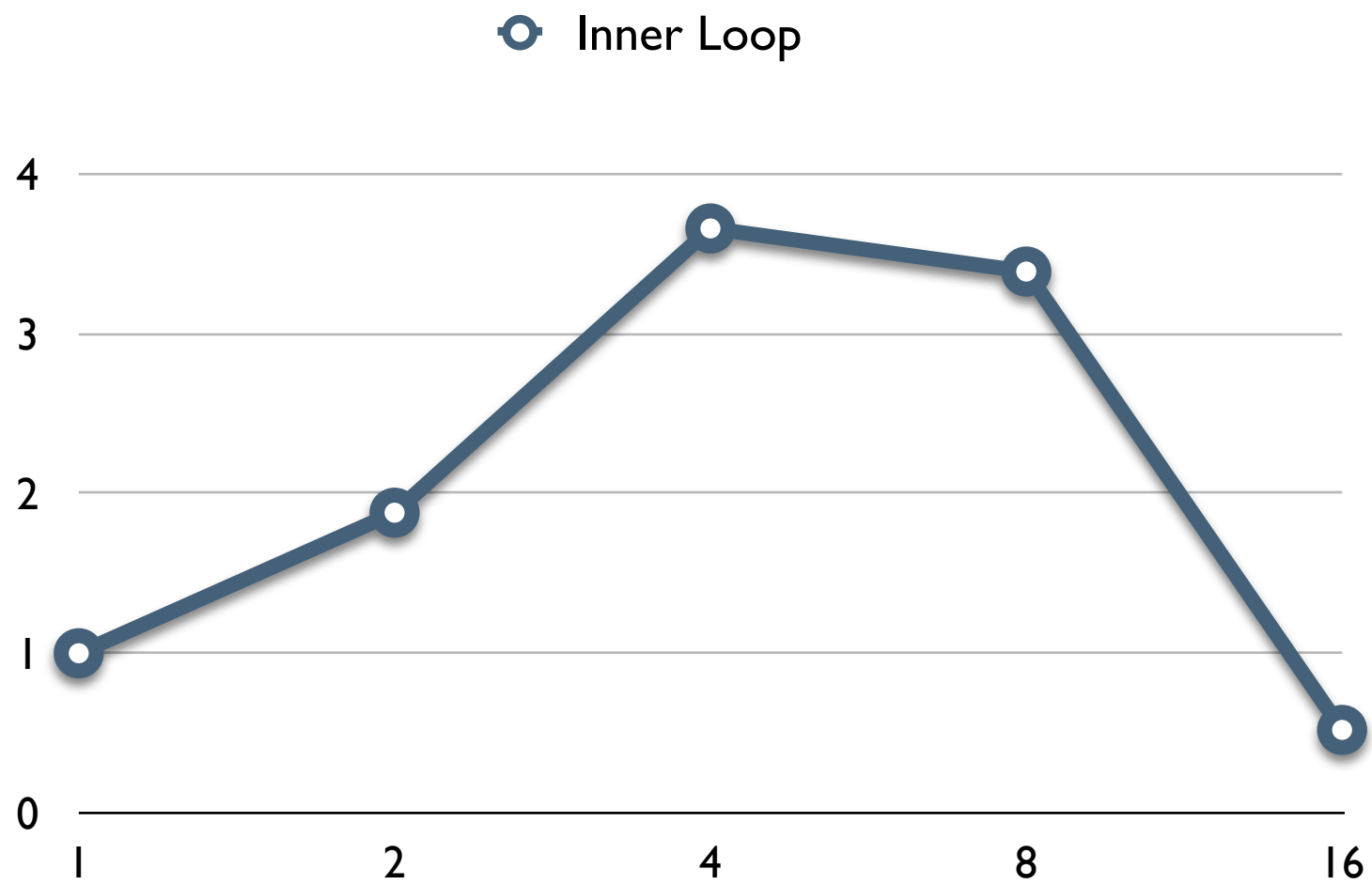
Matrix-vector multiply



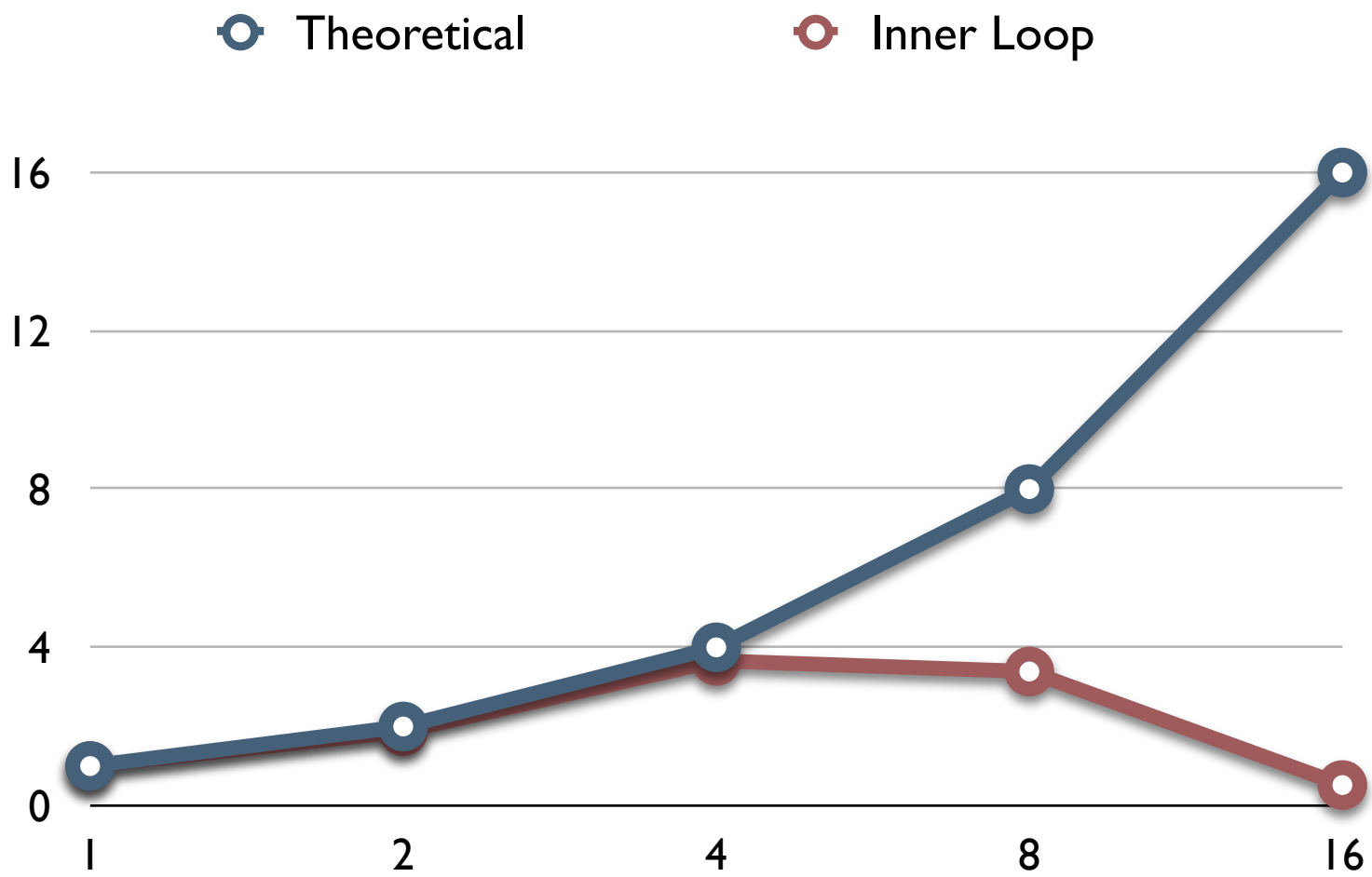
Matrix multiplication

```
!$OMP PARALLEL DO private(rsum,i1,i2,i3)
do i3=1,n
    do i2=1,n
      rsum=0
      do i1=1,n
        rsum=rsum+in1(i1,i3)*in2(i2,i1)
      end do
      out(i2,i3)=rsum
    end do
  end do
  !OMP END PARALLEL DO
```

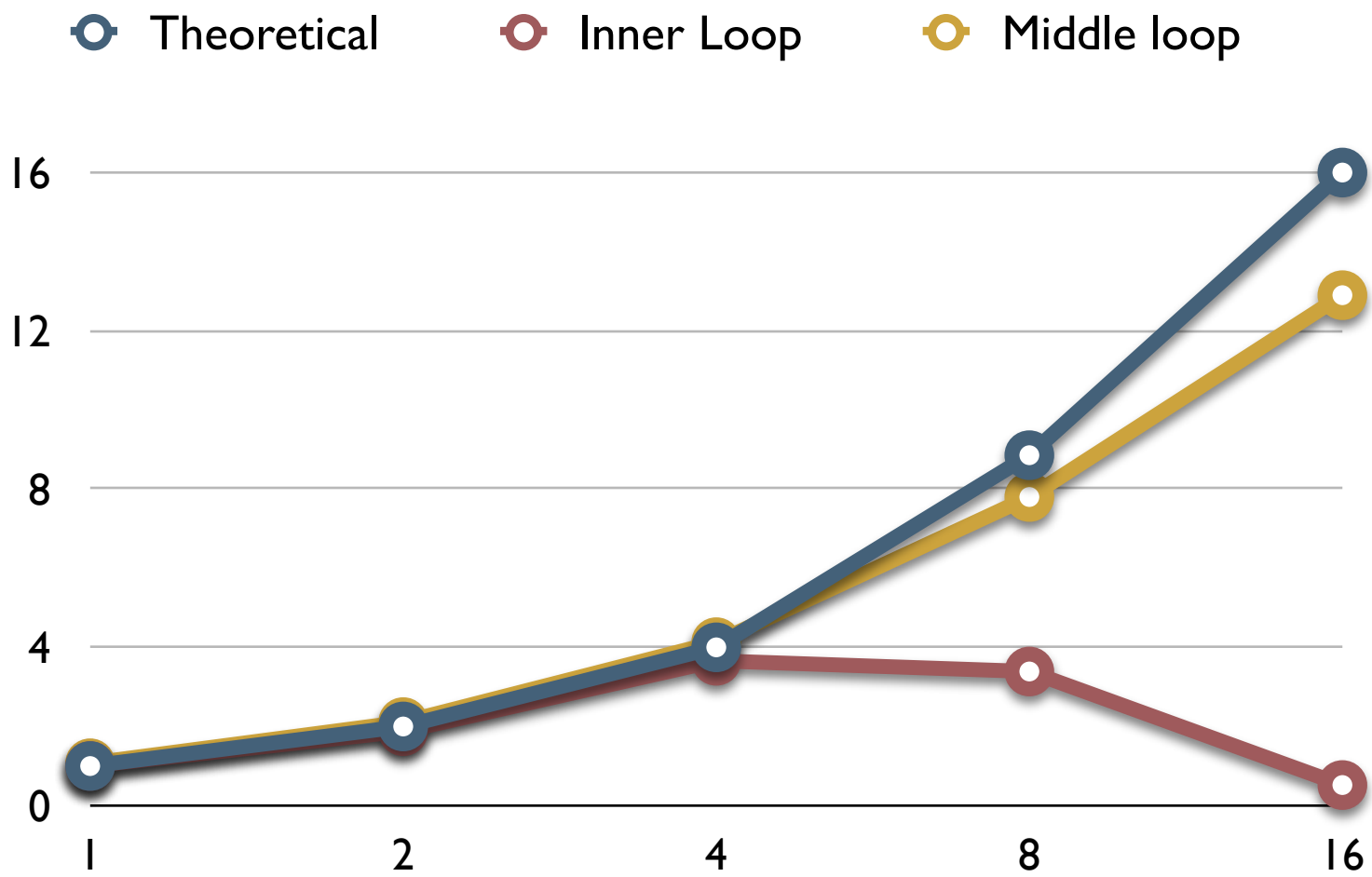
Matrix multiply



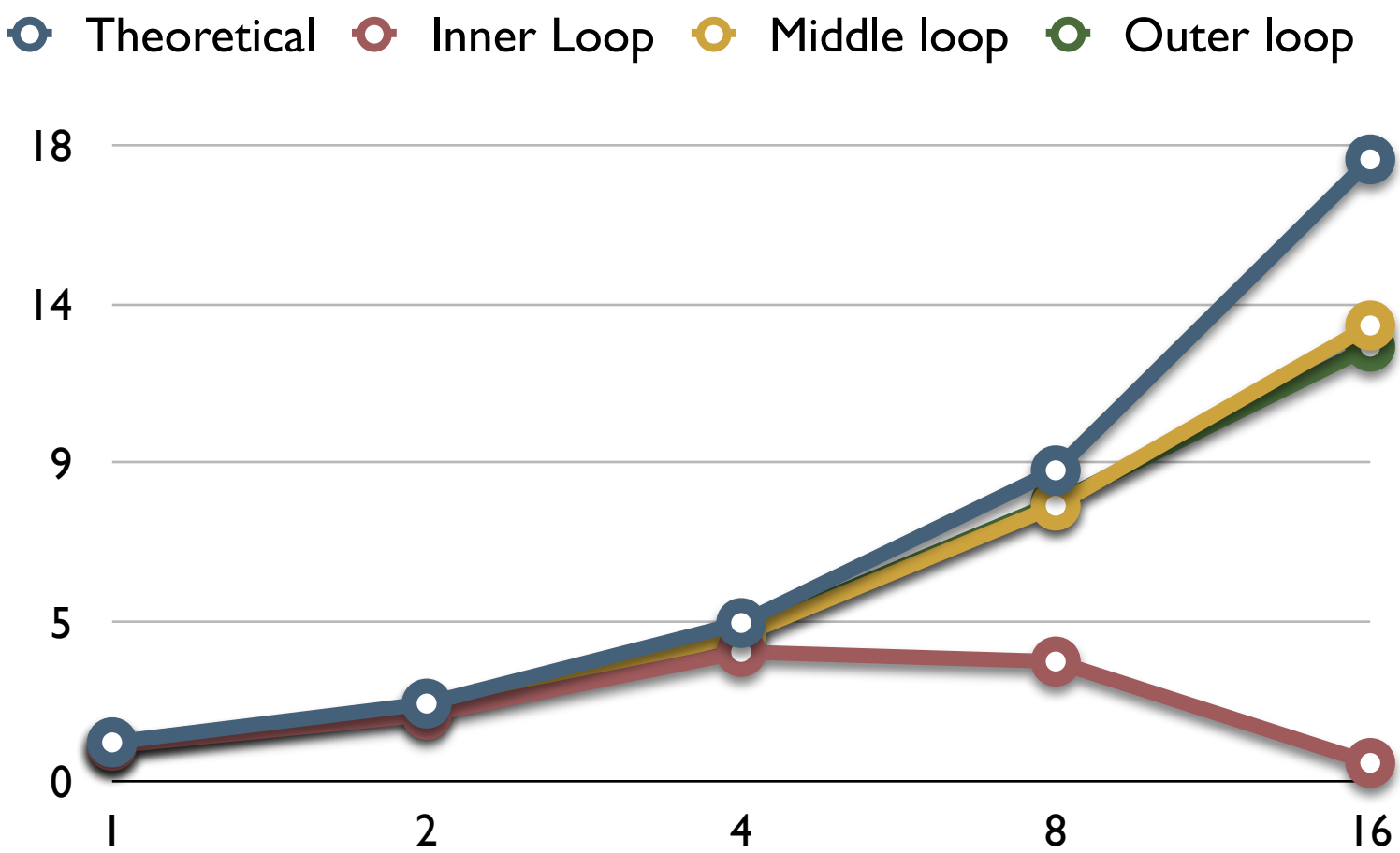
Matrix multiply



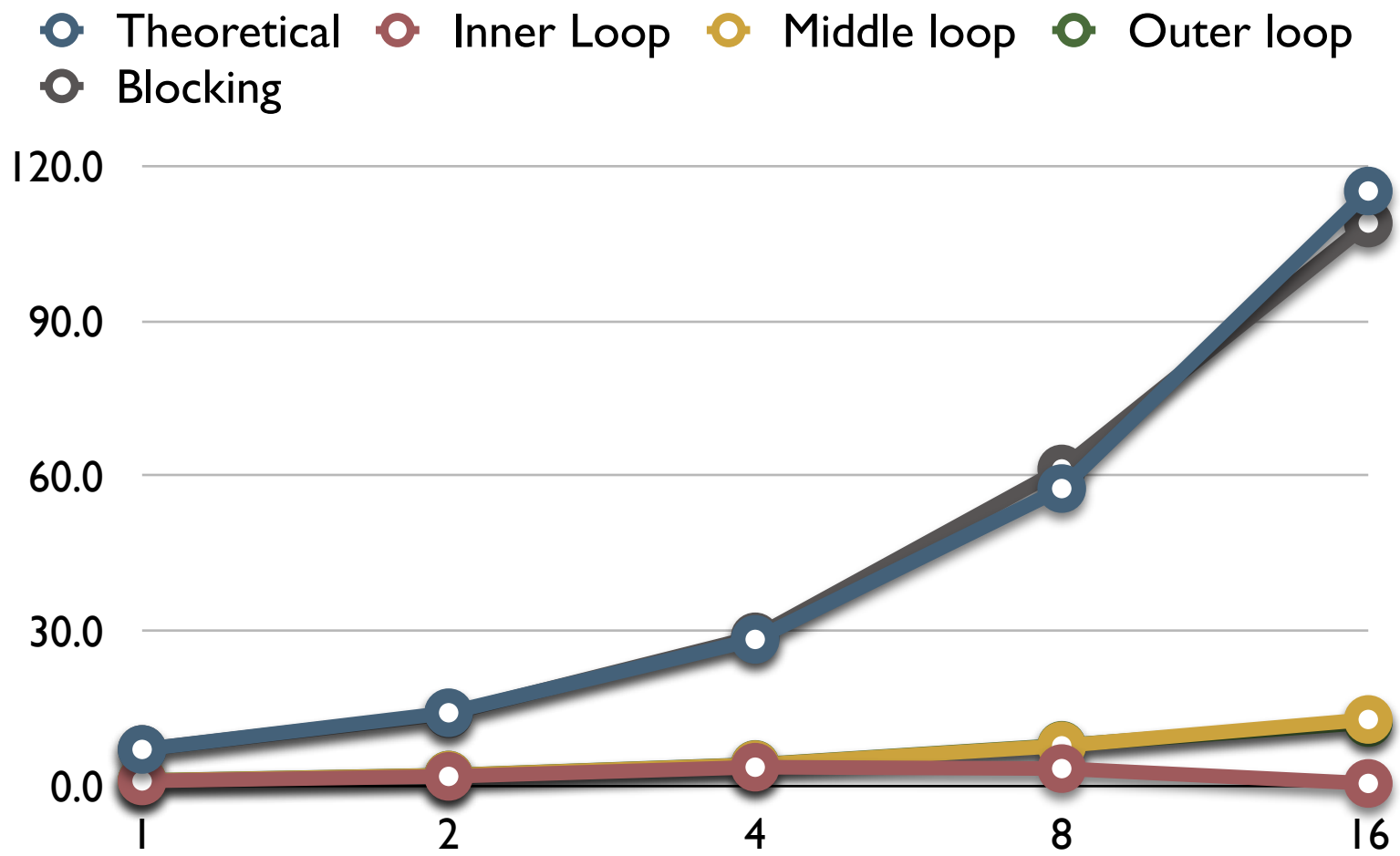
Matrix multiply



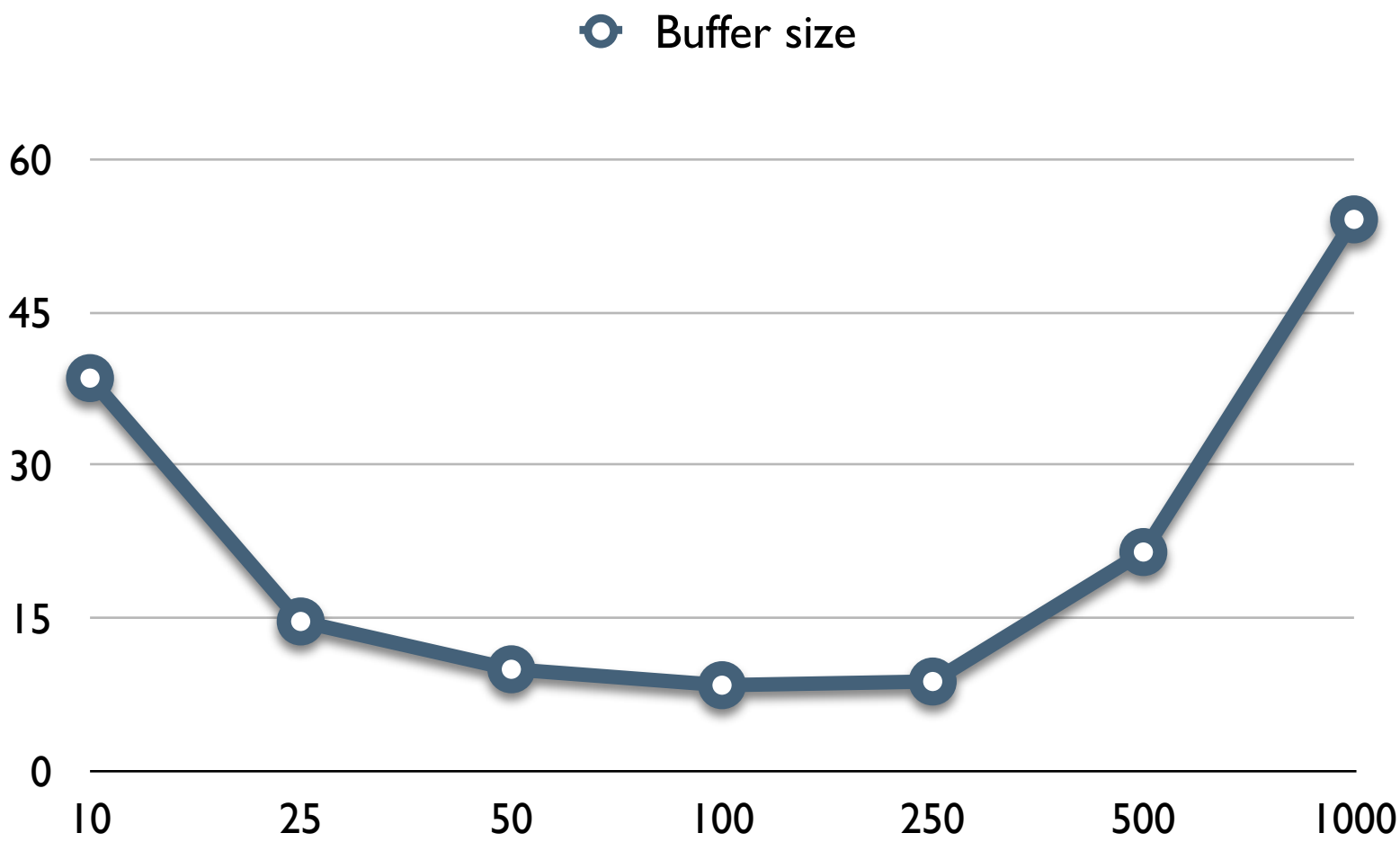
Matrix multiply



Matrix multiply



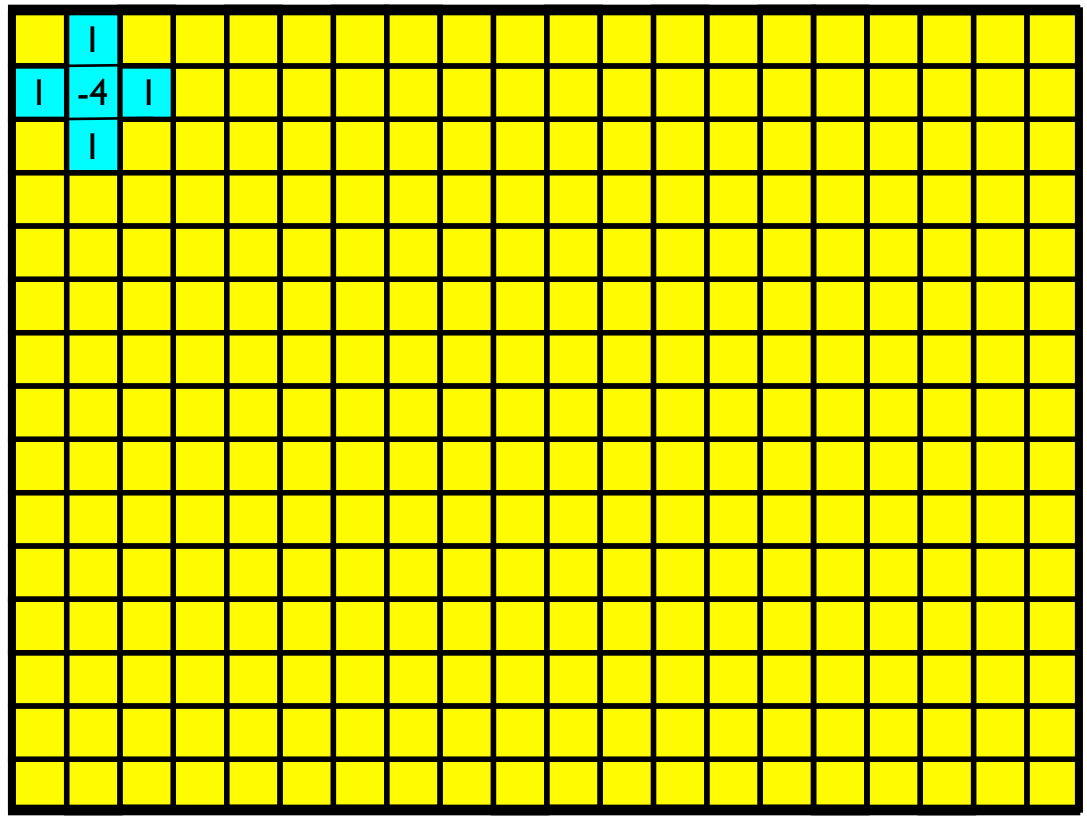
Buffer size



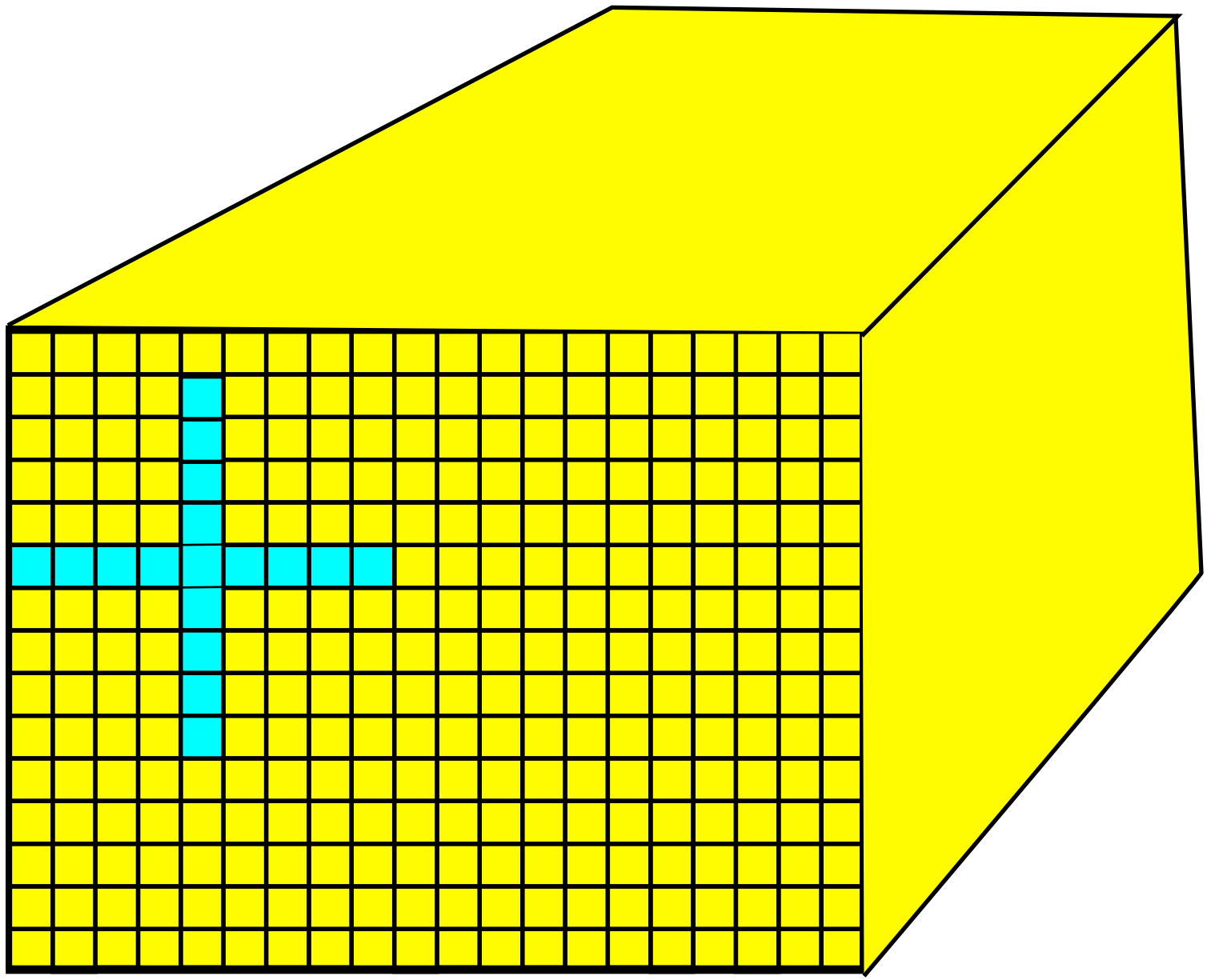
Convolution

```
for(it=0; it < nt; it++{
  do_convolution
  pm=p; p=pp
}
```

```
 for (ix=1; ix<nx-1; ++ix) {
    for (iz=1; iz<nz-1; ++iz) {
      pp[ix][iz] = 2.0*p[ix][iz]-pm[ix][iz] +
       scale*dvv[ix][iz]*( p[ix+1][iz]+ p[ix-1][iz]+
       p[ix][iz+1]+ p[ix][iz-1]-4.0*p[ix][iz])+s[ix][iz]
    }
 }
```



Convolution: Larger stencil and 3-D



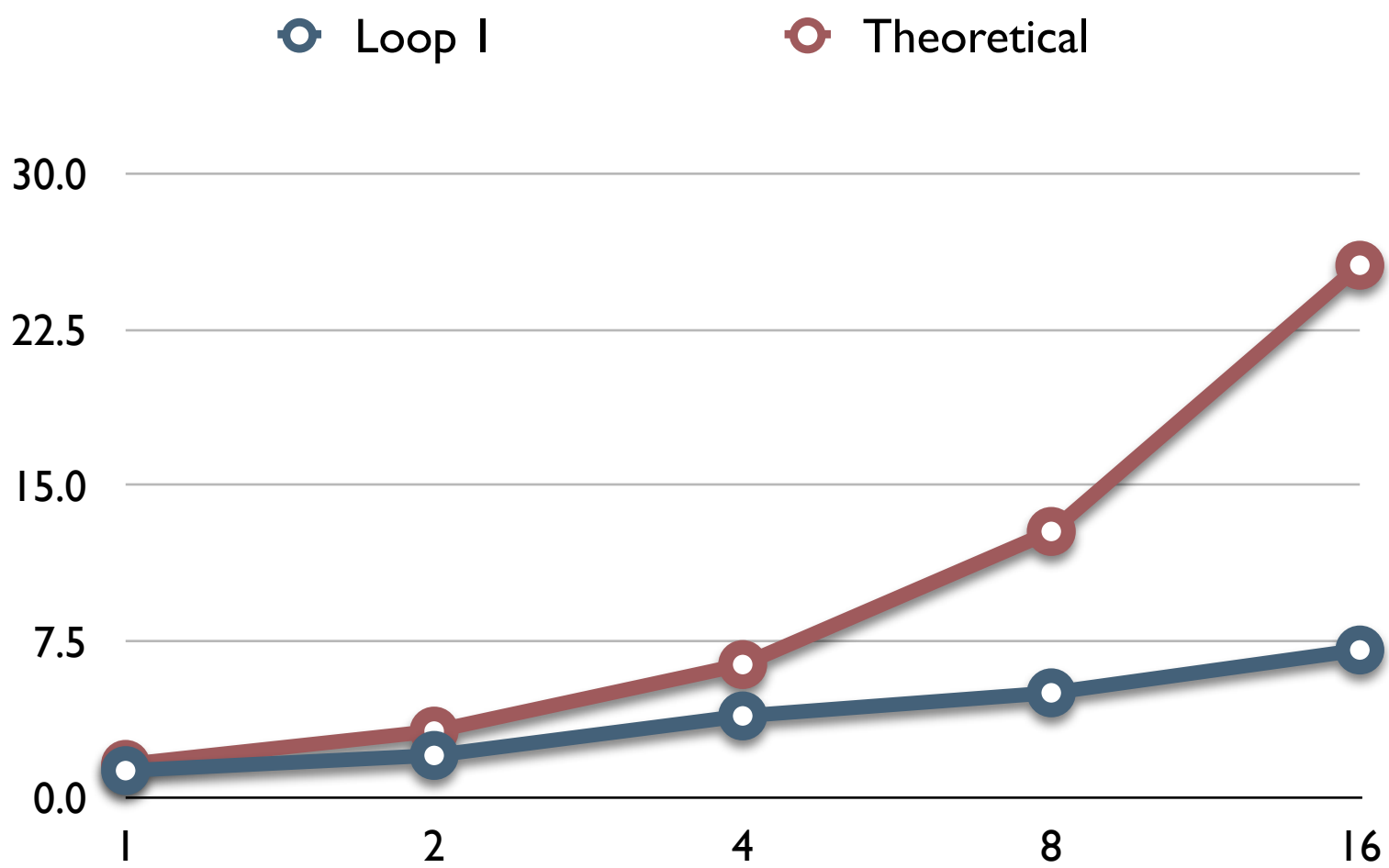
Convolution

```
for(iy=3;iy < ny-4; ++iy){
for (ix=3; ix<nx-4; ++ix) {
   for (iz=3; iz<nz-4; ++iz) {
     pp[iy][ix][iz] = 2.0*p[iy][ix][iz]-pm[iy][ix][iz] +
      scale*dvv[ix][iz]*(
       a3*p[iy-3][ix][iz]+a2*p[iy-2][ix][iz]+a1*p[iy-1][ix][iz]+a0*p[iy][ix][iz]+
      a1*p[iy+1][ix][iz]+a2*p[iy+2][ix][iz]+a3*p[iy+3][ix][iz]+
         a3*p[iy][ix-3][iz]+a2*p[iy][ix-2][iz]+a1*p[iy][ix-1][iz]+a0*p[iy][ix][iz]+
      a1*p[iy][ix+1][iz]+a2*p[iy][ix+2][iz]+a3*p[iy][ix+3][iz]+
      a3*p[iy][ix][iz-3]+a2*p[iy][ix][iz-2]+a1*p[iy][ix][iz-1]+a0*p[iy][ix][iz]+
      a1*p[iy][ix][iz+1]+a2*p[iy][ix][iz+2]+a3*p[iy][ix][iz+3]));
   }
 }
}
```

Convolution

```
for(iy=3;iy < ny-4; ++iy){
for (ix=3; ix<nx-4; ++ix) {
  for (iz=3; iz<nz-4; ++iz) {
    pp[iy][ix][iz] = 2.0*p[iy][ix][iz]-pm[iy][ix][iz] +
     scale*dvv[ix][iz]*(
      a3*p[iy-3][ix][iz]+a2*p[iy-2][ix][iz]+a1*p[iy-1][ix][iz]+a0*p[iy][ix][iz]+
     a1*p[iy+1][ix][iz]+a2*p[iy+2][ix][iz]+a3*p[iy+3][ix][iz]+
        a3*p[iy][ix-3][iz]+a2*p[iy][ix-2][iz]+a1*p[iy][ix-1][iz]+a0*p[iy][ix][iz]+
     a1*p[iy][ix+1][iz]+a2*p[iy][ix+2][iz]+a3*p[iy][ix+3][iz]+
     a3*p[iy][ix][iz-3]+a2*p[iy][ix][iz-2]+a1*p[iy][ix][iz-1]+a0*p[iy][ix][iz]+
     a1*p[iy][ix][iz+1]+a2*p[iy][ix][iz+2]+a3*p[iy][ix][iz+3]));
  }
 }
}
```

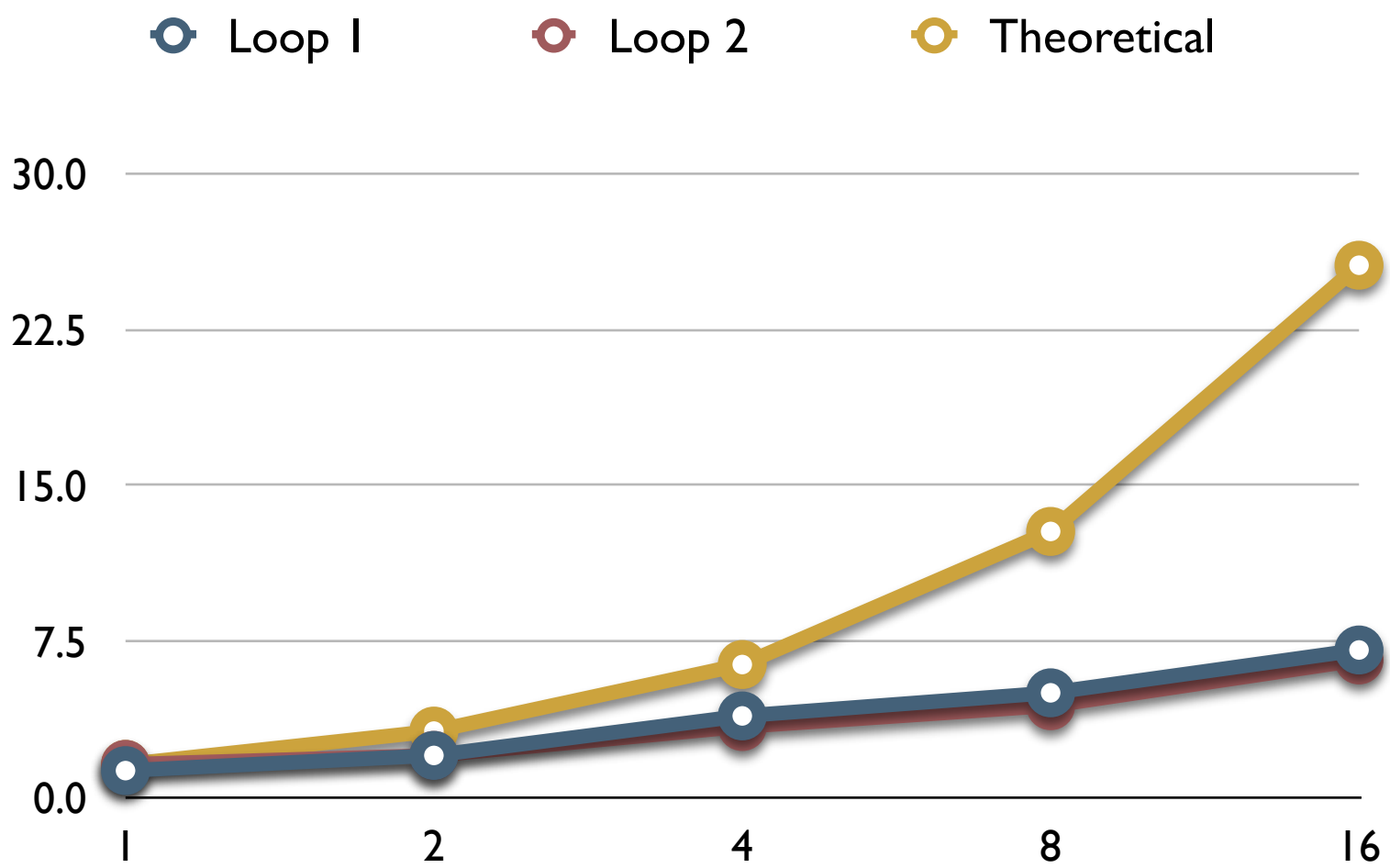
Convolution



Convolution

```
for(iy=3;iy < ny-4; ++iy){
for (ix=3; ix<nx-4; ++ix) {
   for (iz=3; iz<nz-4; ++iz) {
     pp[iy][ix][iz] = 2.0*p[iy][ix][iz]-pm[iy][ix][iz] +
      scale*dvv[ix][iz]*(
       a3*p[iy-3][ix][iz]+a2*p[iy-2][ix][iz]+a1*p[iy-1][ix][iz]+a0*p[iy][ix][iz]+
      a1*p[iy+1][ix][iz]+a2*p[iy+2][ix][iz]+a3*p[iy+3][ix][iz]+
         a3*p[iy][ix-3][iz]+a2*p[iy][ix-2][iz]+a1*p[iy][ix-1][iz]+a0*p[iy][ix][iz]+
      a1*p[iy][ix+1][iz]+a2*p[iy][ix+2][iz]+a3*p[iy][ix+3][iz]+
      a3*p[iy][ix][iz-3]+a2*p[iy][ix][iz-2]+a1*p[iy][ix][iz-1]+a0*p[iy][ix][iz]+
      a1*p[iy][ix][iz+1]+a2*p[iy][ix][iz+2]+a3*p[iy][ix][iz+3]));
   }
 }
}
```

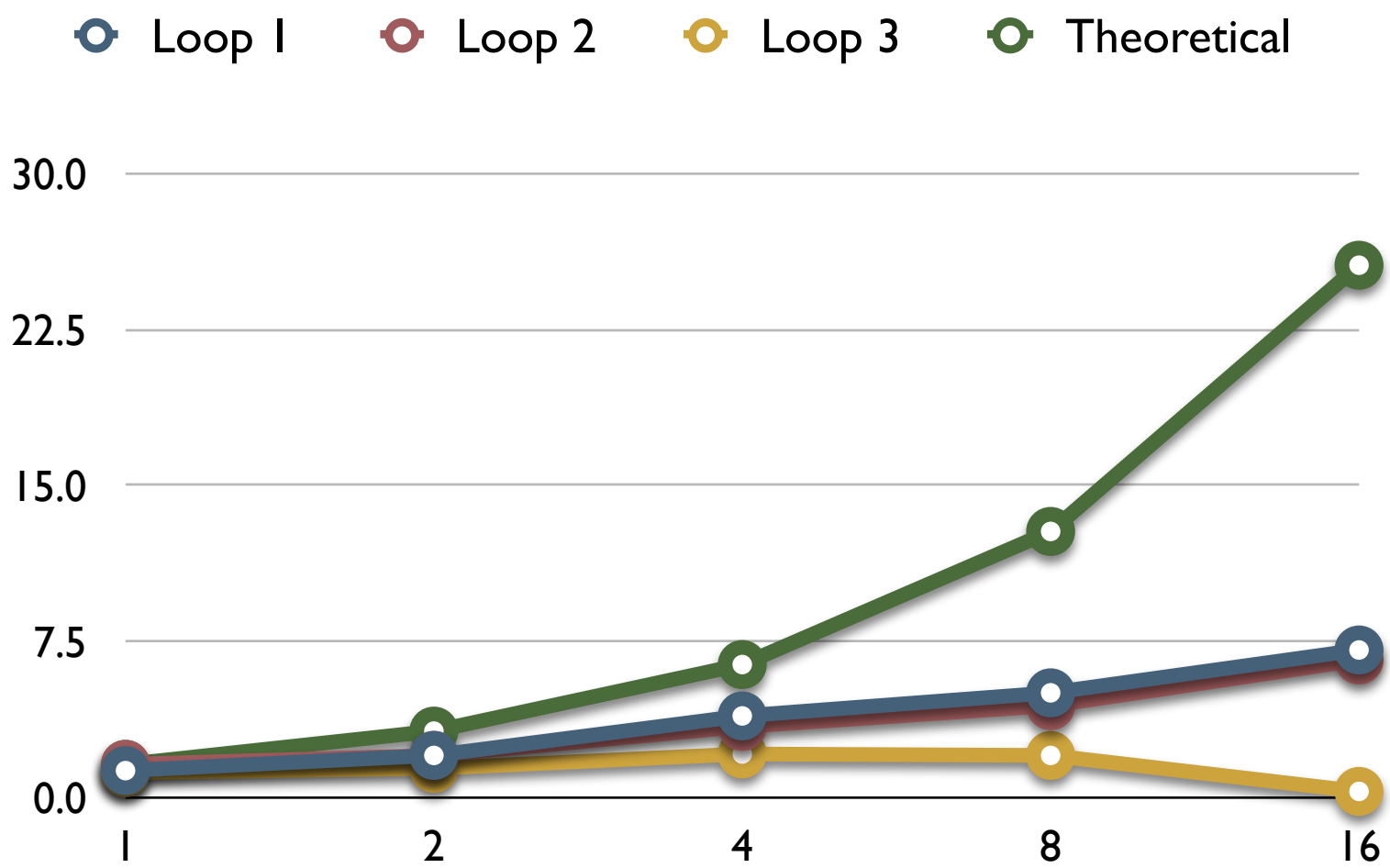
Convolution



Convolution

```
for(iy=3;iy < ny-4; ++iy){
for (ix=3; ix<nx-4; ++ix) {
   for (iz=3; iz<nz-4; ++iz) {
     pp[iy][ix][iz] = 2.0*p[iy][ix][iz]-pm[iy][ix][iz] +
      scale*dvv[ix][iz]*(
       a3*p[iy-3][ix][iz]+a2*p[iy-2][ix][iz]+a1*p[iy-1][ix][iz]+a0*p[iy][ix][iz]+
      a1*p[iy+1][ix][iz]+a2*p[iy+2][ix][iz]+a3*p[iy+3][ix][iz]+
         a3*p[iy][ix-3][iz]+a2*p[iy][ix-2][iz]+a1*p[iy][ix-1][iz]+a0*p[iy][ix][iz]+
      a1*p[iy][ix+1][iz]+a2*p[iy][ix+2][iz]+a3*p[iy][ix+3][iz]+
      a3*p[iy][ix][iz-3]+a2*p[iy][ix][iz-2]+a1*p[iy][ix][iz-1]+a0*p[iy][ix][iz]+
      a1*p[iy][ix][iz+1]+a2*p[iy][ix][iz+2]+a3*p[iy][ix][iz+3]));
   }
 }
}
```

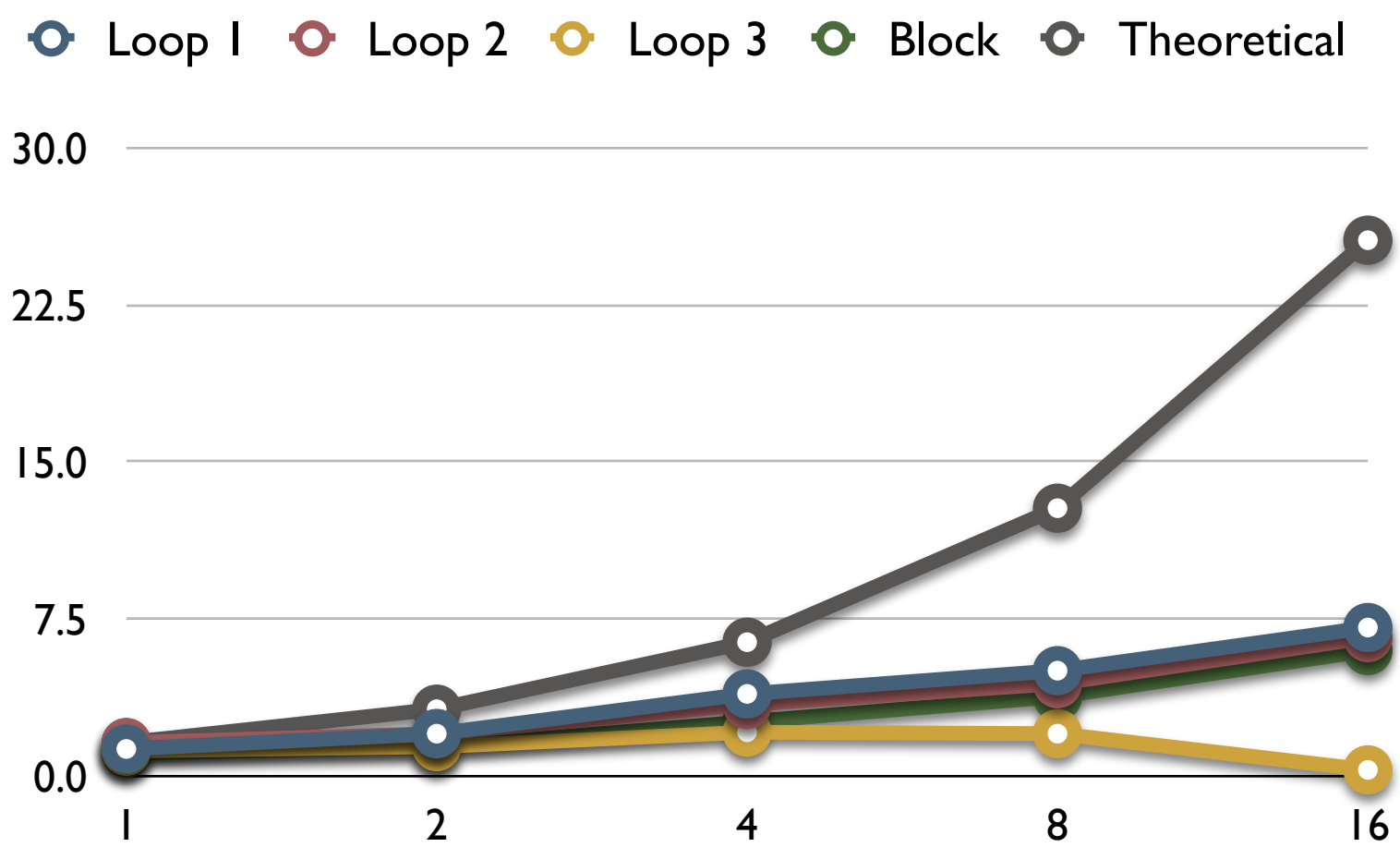
Convolution



Convolution

```
for(iy=3;iy < ny-4; ++iy){
 for (ix=3; ix<nx-4; ++ix) {
   for (iz=3; iz<nz-4; ++iz) {
     pp[iy][ix][iz] = 2.0*p[iy][ix][iz]-pm[iy][ix][iz] +
      scale*dvv[ix][iz]*(
       a3*p[iy-3][ix][iz]+a2*p[iy-2][ix][iz]+a1*p[iy-1][ix][iz]+a0*p[iy][ix][iz]+
      a1*p[iy+1][ix][iz]+a2*p[iy+2][ix][iz]+a3*p[iy+3][ix][iz]+
         a3*p[iy][ix-3][iz]+a2*p[iy][ix-2][iz]+a1*p[iy][ix-1][iz]+a0*p[iy][ix][iz]+
      a1*p[iy][ix+1][iz]+a2*p[iy][ix+2][iz]+a3*p[iy][ix+3][iz]+
      a3*p[iy][ix][iz-3]+a2*p[iy][ix][iz-2]+a1*p[iy][ix][iz-1]+a0*p[iy][ix][iz]+
      a1*p[iy][ix][iz+1]+a2*p[iy][ix][iz+2]+a3*p[iy][ix][iz+3]));
   }
 }
}
```

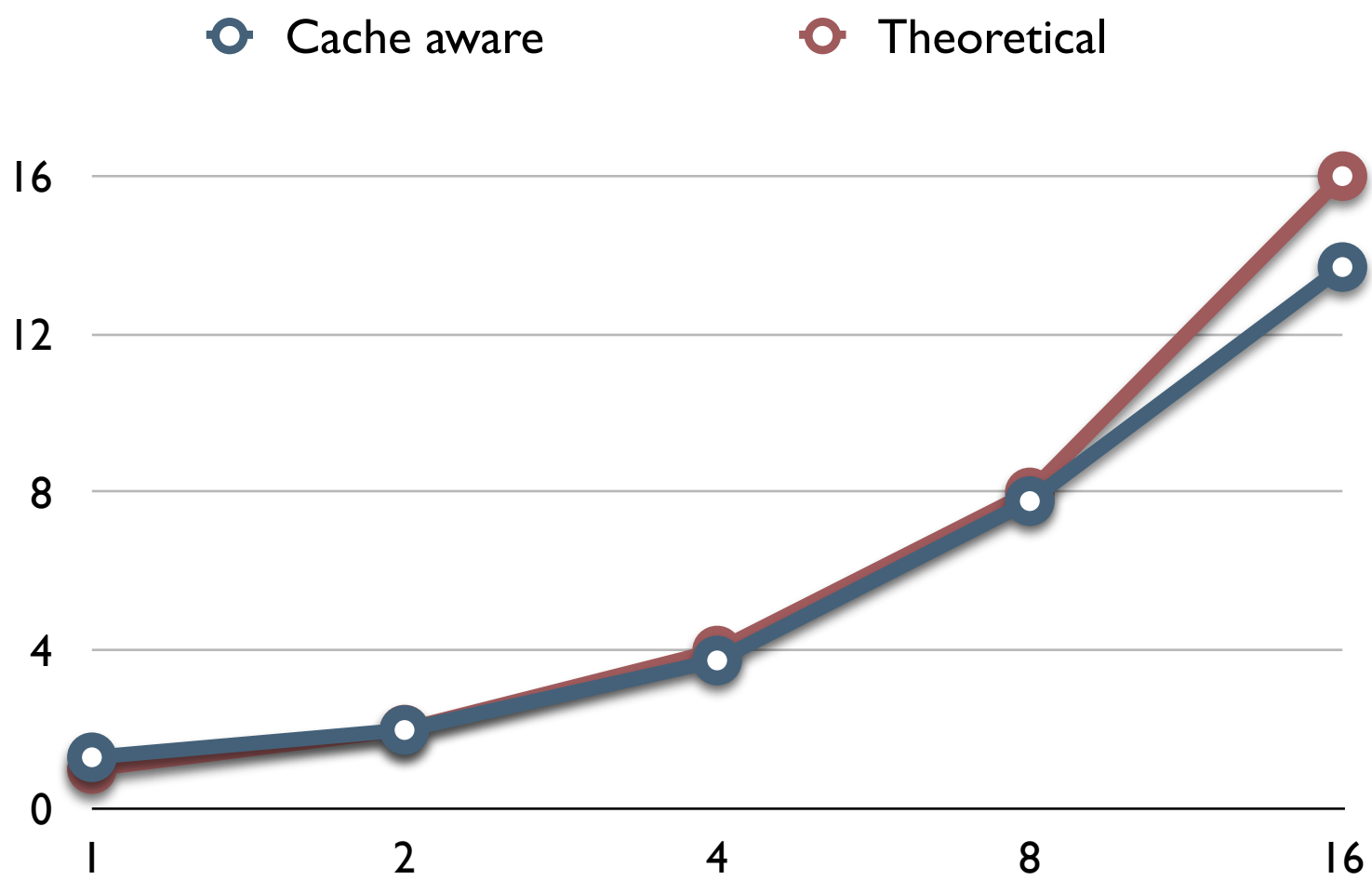
Convolution



Convolution

```
for(iy=3;iy < ny-4; ++iy){
for (ix=3; ix<nx-4; ++ix) {
  for (iz=3; iz<nz-4; ++iz) {
    pp[iy][ix][iz] = 2.0*p[iy][ix][iz]-pm[iy][ix][iz] +
     scale*dvv[ix][iz]*(
      a3*p[iy-3][ix][iz]+a2*p[iy-2][ix][iz]+a1*p[iy-1][ix][iz]+a0*p[iy][ix][iz]+
     a1*p[iy+1][ix][iz]+a2*p[iy+2][ix][iz]+a3*p[iy+3][ix][iz]+
        a3*p[iy][ix-3][iz]+a2*p[iy][ix-2][iz]+a1*p[iy][ix-1][iz]+a0*p[iy][ix][iz]+
     a1*p[iy][ix+1][iz]+a2*p[iy][ix+2][iz]+a3*p[iy][ix+3][iz]+
     a3*p[iy][ix][iz-3]+a2*p[iy][ix][iz-2]+a1*p[iy][ix][iz-1]+a0*p[iy][ix][iz]+
     a1*p[iy][ix][iz+1]+a2*p[iy][ix][iz+2]+a3*p[iy][ix][iz+3]));
  }
 }
}
```

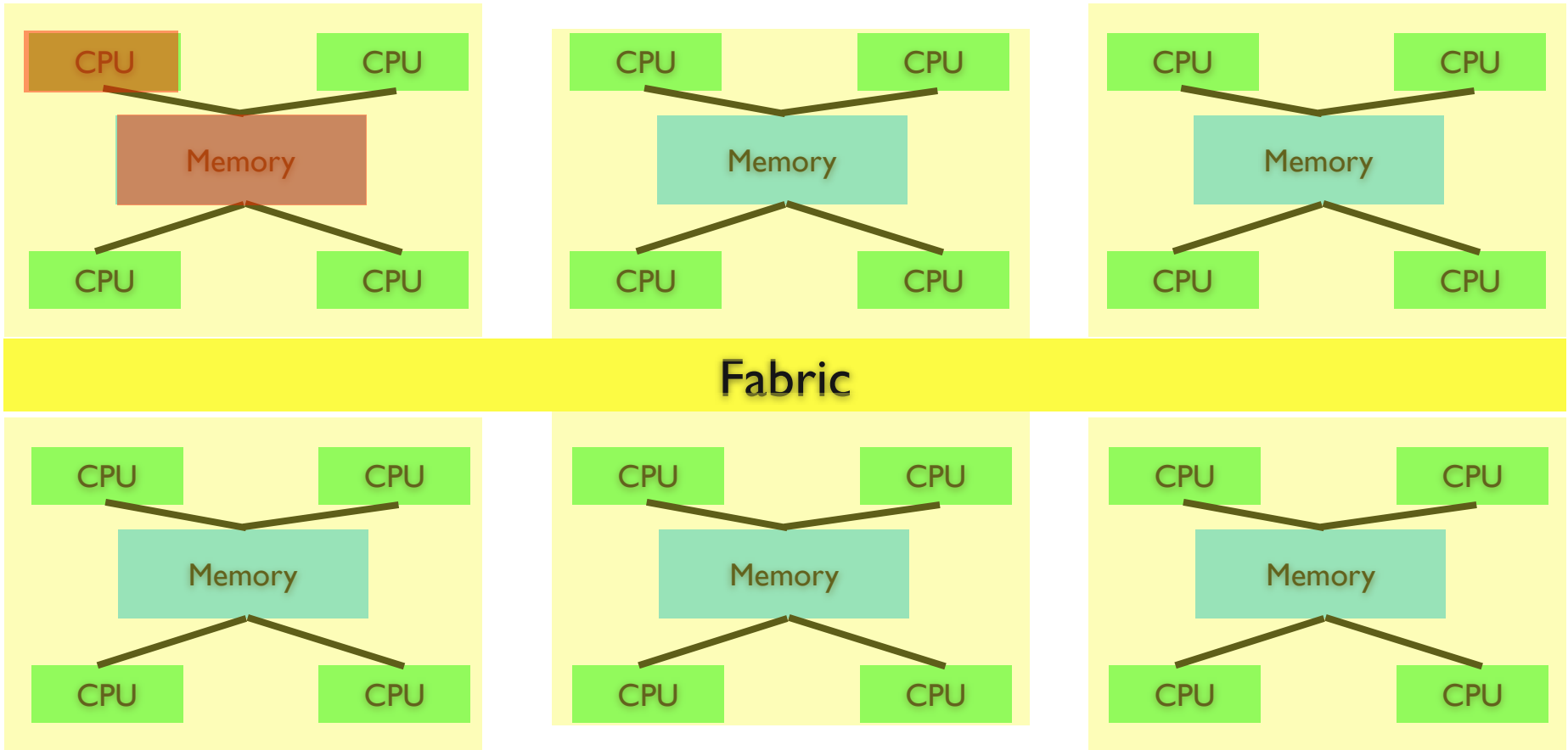
Convolution



OpenMP Arrays

- Arrays are locally allocated
- Input arrays should generally be left unchanged
- Generally don’t allocate/deallocate in parallel region

CPU will allocate memory locally



Depending on your NUMA machine

```
void main(int argc, char **argv){
.
.

nth=1; //For serial case
#pragma omp parallel
nth=omp_get_num_threads();
#pragma omp end parallel

in=(float**)alloc_2d_float(n1,n2);
work=(float**)alloc_2d_float(n1,nth);
out=(float**) alloc_2d_float(n1,n2);

#pragma omp parallel for
for(i=0; i < n2; i++){
  ith=omp_get_thread_num();
  serial_work(in,out,work,i,ith);
}
```

```
void serial_work(float **in, float **out, float **work, int i,
int ith){
float alpha;
.
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[ith][j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[j]...)
  }
}
```

Depending on your NUMA machine

```
void main(int argc, char **argv){
.
.

nth=1; //For serial case
#pragma omp parallel
nth=omp_get_num_threads();
#pragma omp end parallel

in=(float**)alloc_2d_float(n1,n2);
work=(float**)malloc(sizeof(float*)*nth);
out=(float**) alloc_2d_float(n1,n2);

#pragma omp parallel
{
ith=omp_get_thread_num();
work[ith]=(float*)malloc(sizeof(float)*n1);
#pragma omp for private(i)
for(i=0; i < n2; i++){
  serial_work(in,out,work,i,ith);
}
}
```

```
void serial_work(float **in, float **out, float **work, int i,
int ith){
float alpha;
.
  for(j=0; j < n1; j++){
    .
    .
    alpha=func(in[i][j]....)
    work[ith][j]=func(in[i][j].....)
    .
    .
    out[i][j]=func(work[j]...)
  }
}
```

Memory allocate locally

OpenMP essentials

- Begin and end parallel block
- Private vs public variables
- Data parallelism with for/do
- Intermediate arrays expand dimensionality by nthreads

OpenMP: What to remember when parallelizing

- Coarser grain the better
- Limit memory contention (most compute per word)
- Parallelism is often limited by memory bus

Nested parallelism

```
program test_nested
integer, external :: omp_get_thread_num
integer :: main

write(0,*) "YEAH"
!$OMP PARALLEL private(main)
main=omp_get_thread_num()
write(0,*) "ONE",main
!$OMP PARALLEL
write(0,*)  "TWO",omp_get_thread_num(),main
!$OMP END PARALLEL
!$OMP END PARALLEL

end program
```

Nested parallelism

```
program test_nested
integer, external :: omp_get_thread_num
integer :: main

write(0,*) "YEAH"
!$OMP PARALLEL private(main)
main=omp_get_thread_num()
write(0,*) "ONE",main
!$OMP PARALLEL
write(0,*)  "TWO",omp_get_thread_num(),main
!$OMP END PARALLEL
!$OMP END PARALLEL

end program
```

What is going to be the result?

Nested parallelism: Default result

```
program test_nested
integer, external :: omp_get_thread_num
integer :: main

write(0,*) "YEAH"
!$OMP PARALLEL private(main)
main=omp_get_thread_num()
write(0,*) "ONE",main
!$OMP PARALLEL
write(0,*)  "TWO",omp_get_thread_num(),main
!$OMP END PARALLEL
!$OMP END PARALLEL

end program
```

```
YEAH
 ONE           0
TWO            0           0
 ONE           2
TWO            0           2
 ONE           1
 ONE           3
TWO            0           3
TWO            0           1
```

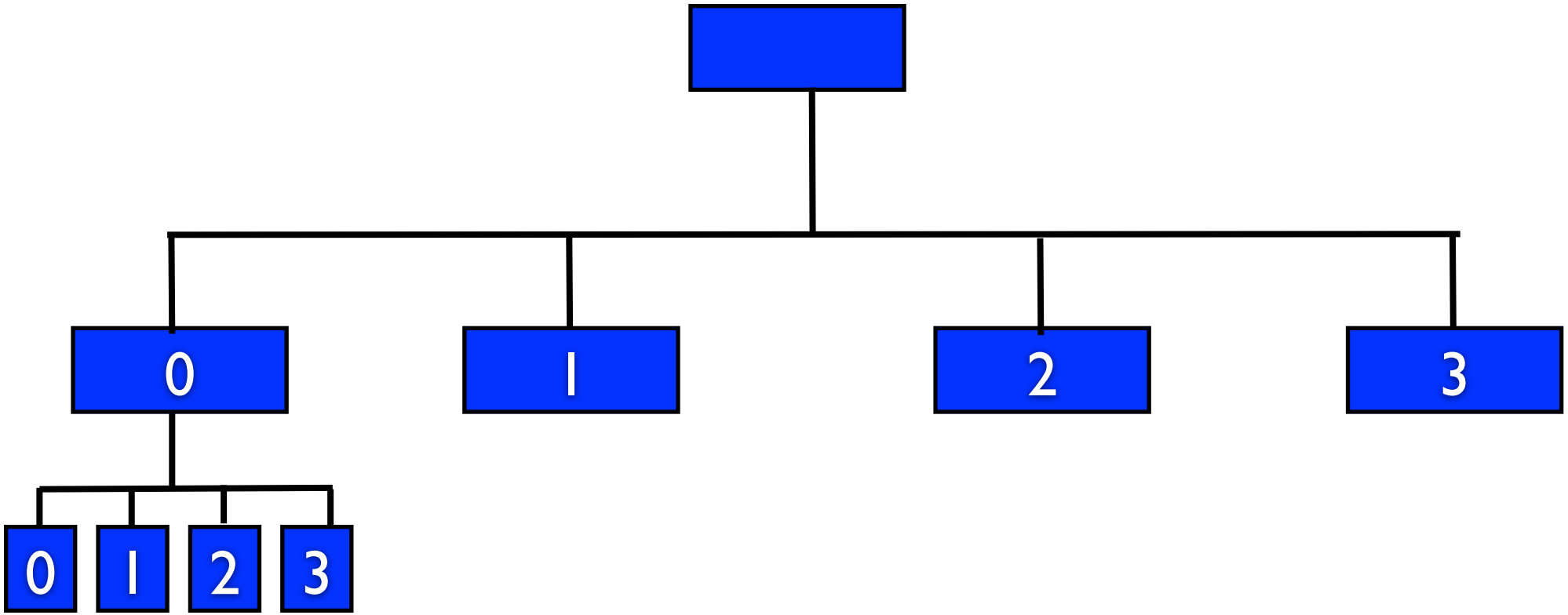
Nested parallelism

setenv OMP_NESTED TRUE

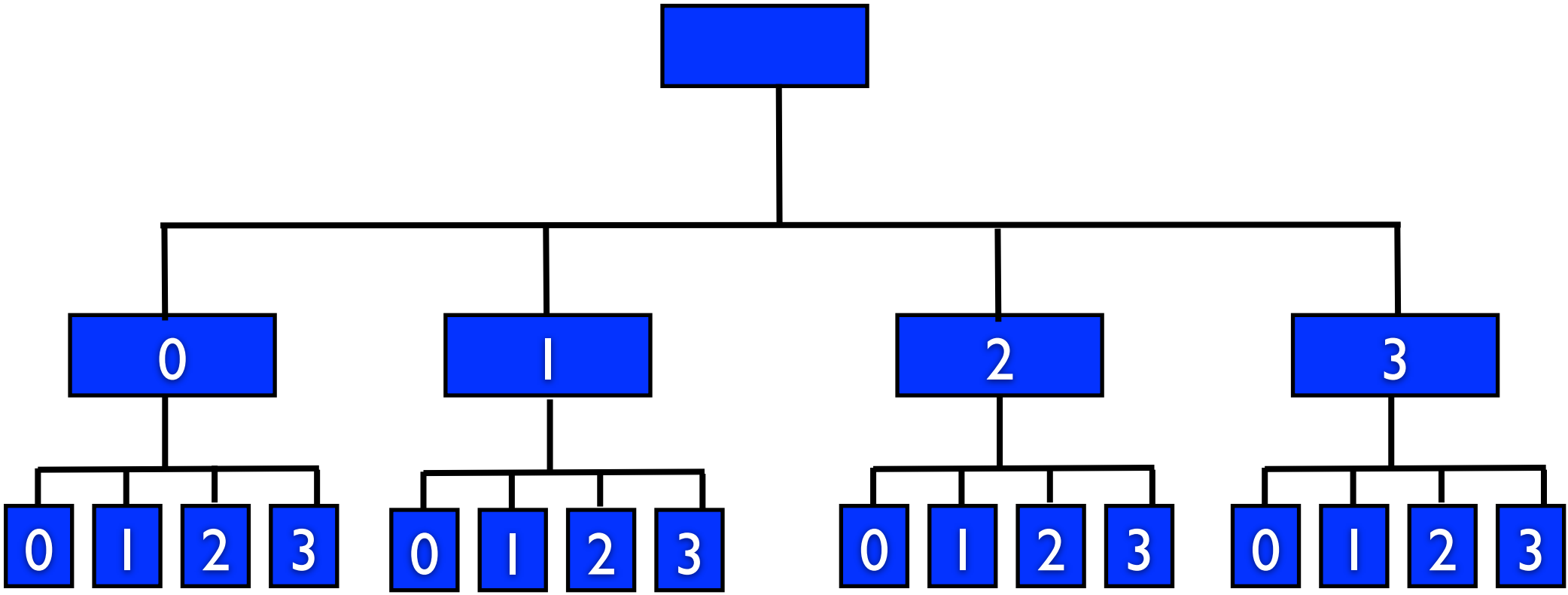
```
program test_nested
integer, external :: omp_get_thread_num
integer :: main

write(0,*) "YEAH"
!$OMP PARALLEL private(main)
main=omp_get_thread_num()
write(0,*) "ONE",main
!$OMP PARALLEL
write(0,*)  "TWO",omp_get_thread_num(),main
!$OMP END PARALLEL
!$OMP END PARALLEL

end program
```

```
 YEAH
 ONE           0
 TWO           0           0
 TWO           3           0
 ONE           1
 TWO           0           1
 ONE           2
 TWO           1           0
 ONE           3
 TWO           2           0
 TWO           3           1
 TWO           2           1
 TWO           0           2
 TWO           0           3
 TWO           1           2
 TWO           3           2
 TWO           2           2
 TWO           3           3
 TWO           2           3
 TWO           1           1
 TWO           1           3
```

Nested parallelism: 1



Nested parallelism: 2



Thread parallelism

```
program thread_parallelism
integer :: i
integer, external :: omp_get_thread_num
real :: buf(100)

!$OMP PARALLEL
do i=1,100
!$OMP TASK
  buf(i)=i
  write(0,*) "CHECK THIS",i,omp_get_thread_num()
!$OMP END TASK
end do
!$OMP END PARALLEL

end program
```

```
CHECK THIS        88          3
CHECK THIS        89          3
CHECK THIS        90          3
CHECK THIS        91          3
CHECK THIS        92          3
CHECK THIS        93          3
CHECK THIS        94          3
CHECK THIS        95          3
CHECK THIS        96          3
CHECK THIS        98          3
CHECK THIS        99          3
CHECK THIS        97          0
CHECK THIS        86          1
CHECK THIS       100          2
```

Thread parallelism

- Create a series of tasks
- Tasks could be executed immediately or one resources are available
- Useful for
  - Consumer/producer
  - Recursive
  - Unbounded loops

Task scoping

- shared
- private
- firstprivate - default for all non-shared variables
  - data is captured at creation
- default(shared|none)

Thread parallelism

```
int a ;
void foo ( ) {
int b , c ;
#pragma omp parallel shared ( b )
#pragma omp parallel private ( b )
{
int d ;
#pragma omp task
{
int e ;
a =
b =
c =
d =
e =
}}}
```

Thread parallelism

```
int a ;
void foo ( ) {
int b , c ;
#pragma omp parallel shared ( b )
#pragma omp parallel private ( b )
{
int d ;
#pragma omp task
{
int e ;
a = shared
b =
c =
d =
e =
} } }
```

Thread parallelism

```
int a ;
void foo ( ) {
int b , c ;
#pragma omp parallel shared ( b )
#pragma omp parallel private ( b )
{
int d ;
#pragma omp task
{
int e ;
a = shared
b = firstprivate
c =
d =
e =
} } }
```

Thread parallelism

```
int a ;
void foo ( ) {
int b , c ;
#pragma omp parallel shared ( b )
#pragma omp parallel private ( b )
{
int d ;
#pragma omp task
{
int e ;
a = shared
b = firstprivate
c = shared
d =
e =
} } }
```

Thread parallelism

```
int a ;
void foo ( ) {
int b , c ;
#pragma omp parallel shared ( b )
#pragma omp parallel private ( b )
{
int d ;
#pragma omp task
{
int e ;
a = shared
b = firstprivate
c = shared
d = firstprivate
e =
} } }
```

Thread parallelism

```
int a ;
void foo ( ) {
int b , c ;
#pragma omp parallel shared ( b )
#pragma omp parallel private ( b )
{
int d ;
#pragma omp task
{
int e ;
a = shared
b = firstprivate
c = shared
d = firstprivate
e = private
} } }
```

OpenMP tasks

- Barrier

  ```
  !$OMP TASKWAIT
  ```

- if statements

Nested parallelism

setenv OMP_NESTED TRUE

```
program nested_two
integer, external :: omp_get_num_threads
integer, external :: omp_get_thread_num
integer :: main,nlocal,ntot

ntot=omp_get_num_threads()
write(0,*) "YEAH"
call omp_set_num_threads(2)
!$OMP PARALLEL private(main)
main=omp_get_thread_num()
call omp_set_num_threads(ntot/2)
write(0,*) "ONE",main
!$OMP PARALLEL
write(0,*)  "TWO",omp_get_thread_num(),main
!$OMP END PARALLEL
!$OMP END PARALLEL
end program
```

```
 YEAH
 ONE           0
 TWO           0           0
 ONE           1
 TWO           0           1
```