OpenMP 4 - What's New?

SciNet Developer Seminar

Ramses van Zon

September 25, 2013

Intro to OpenMP

- For shared memory systems.
- Add parallelism to functioning serial code.
- For C, C++ and Fortran
- http://openmp.org

- Compiler/run-time does a lot of work for you
- Divides up work
- You tell it how to use variables, and what to parallelize.
- Works by adding compiler directives to code.

OpenMP.org

openmp.org/wp/

OpenMP

THE OPENMP® API SPECIFICATION FOR PARALLEL

Subscribe to the News Feed

»OpenMP Specifications

»About the OpenMP ARB
»Frequently Asked Questions
»Compilers
»Resources
»Who's Using OpenMP?
»Press Releases

»Discussion Forums

Events
»*Public OpenMP Calendar*

Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Follow @OpenMP_ARB

OpenMP News

»Recently Published Articles on OpenMP

Admin Magazine/HPC
http://www.admin-magazine.com/HPC/Articles/HPC-Software-Road-Gets-a-Bit-Sm

The Clang/LLVM compiler now supports OpenMP 3.1
http://clang-omp.github.io/

Linux Journal / Advanced OpenMP
http://ow.ly/2z2A8p

In Online Journal *Embedded*
A portable OpenMP runtime library based on MCA APIs for embedded systems – Part
http://ow.ly/2yUR1s

ACM Digital Library
Portable mapping of OpenMP to Multicore embedded systems using MCA APIs
http://dl.acm.org/citation.cfm?id=2465569&bnc=1

Posted on September 17, 2013

»University of Houston Joins the OpenMP Effort

The University of Houston (UH) has joined the OpenMP Consortium, a group of leading and software vendors and research organizations creating the standard for the most po memory parallel programming model in use today.

"We are excited to join the OpenMP family as an academic member", says Barbara Ch Professor of Computer Science and director of the Center for Advanced Computing an Systems (CACDS) at UH. "Through cOMPunity, we have been involved since the begi this new membership, we will continue to engage with the OpenMP community."

"This is a great step forward for the UH HPCTools research group. We look forward to

Quick Example - C

```c
/* example1.c */
int main()
{
  int i,sum;
  sum=0;


  for (i=0; i<101; i++)
    sum+=i;

  return sum-5050;
}
```

⇒

```c
/* example1.c */
int main()
{
  int i,sum;
  sum=0;
 #pragma omp parallel
 #pragma omp for reduction(+:sum)
  for (i=0; i<101; i++)
    sum+=i;

  return sum-5050;
}
```

```
> $CC example1.c
> ./a.out
```

```
> $CC example1.c -fopenmp
> export OMP_NUM_THREADS=8
> ./a.out
```

SciNet compute • calcul CANADA

Quick Example - Fortran

```
program example1
  integer i,sum
  sum=0


  do i=1,100
     sum=sum+i
  end do

  print *, sum-5050;
end program example1
```

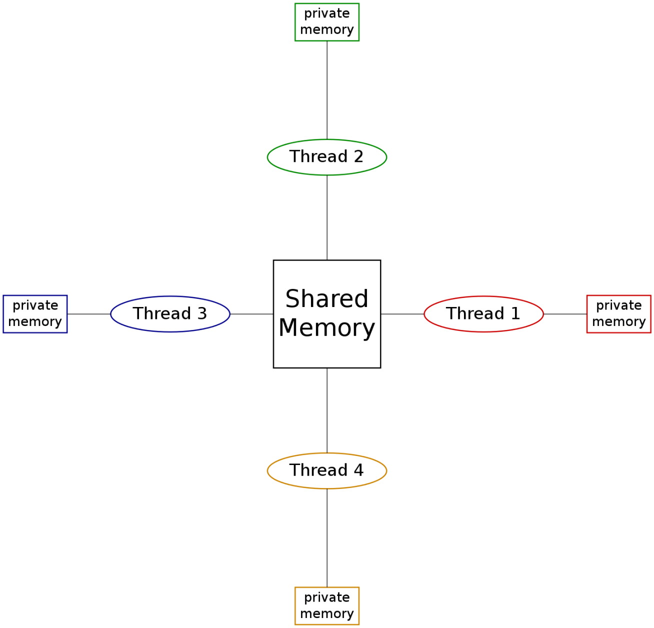
⇒

```
program example1
  integer i,sum
  sum=0
 !$omp parallel
 !$omp do reduction(+:sum)
  do i=1,100
     sum=sum+i
  end do
 !$omp end parallel
  print *, sum-5050;
end program example1
```

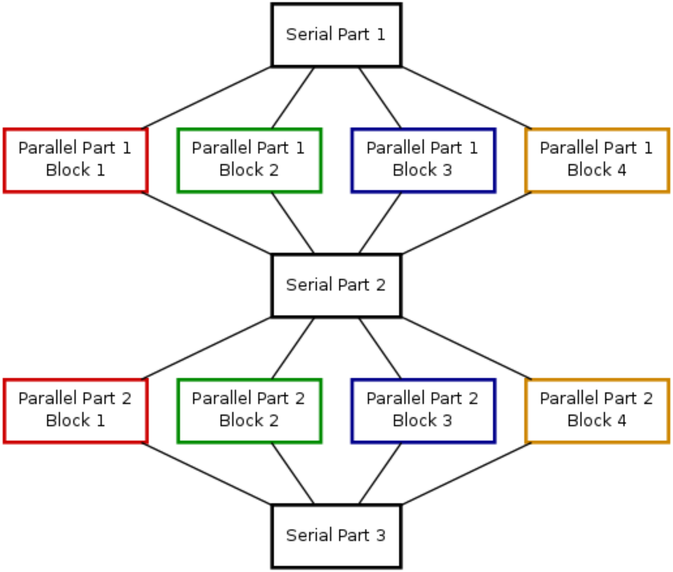
```
> $FC example1.f90
```

```
> $FC example1.f90 -fopenmp
```

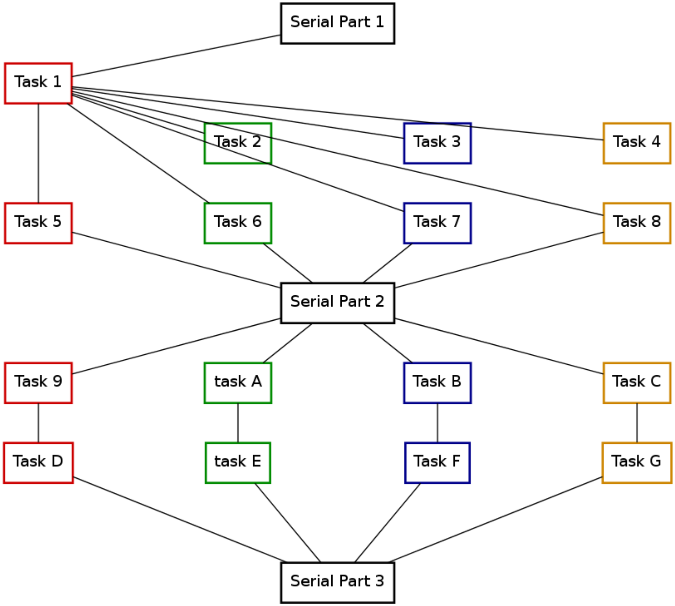
Memory Model in OpenMP (3.1)



Execution Model in OpenMP



Execution Model in OpenMP with Tasks



Existing Features (OpenMP 3.1)

1. Create threads with shared and private memory;
2. Parallel sections and loops;
3. Different work scheduling algorithms for load balancing loops;
4. Lock, critical and atomic operations to avoid race conditions;
5. Combining results from different threads;
6. Nested parallelism;
7. Generating task to be executed by threads.

Supported by GCC, Intel, PGI and IBM XL compilers.

Introducing OpenMP 4.0

- Released July 2013, OpenMP 4.0 is an API *specification*.
- As usual with standards, it's a mix of features that are commonly implemented in another form and ones that have never been implemented.
- As a result, compiler support varies. E.g. Intel compilers v. 14.0 good at offloading to phi, gcc has more task support.
- OpenMP 4.0 is 248 page document (without appendices) (OpenMP 1 C/C++ or Fortran was $\approx$ 40 pages)
- No examples in this specification, no summary card either.
- But it has a lot of new features. . .

New Features in OpenMP 4.0

1. Support for compute devices
2. SIMD constructs
3. Task enhancements
4. Thread affinity
5. Other improvements

1. Support for Compute Devices

- Effort to support a wide variety of compute devices:

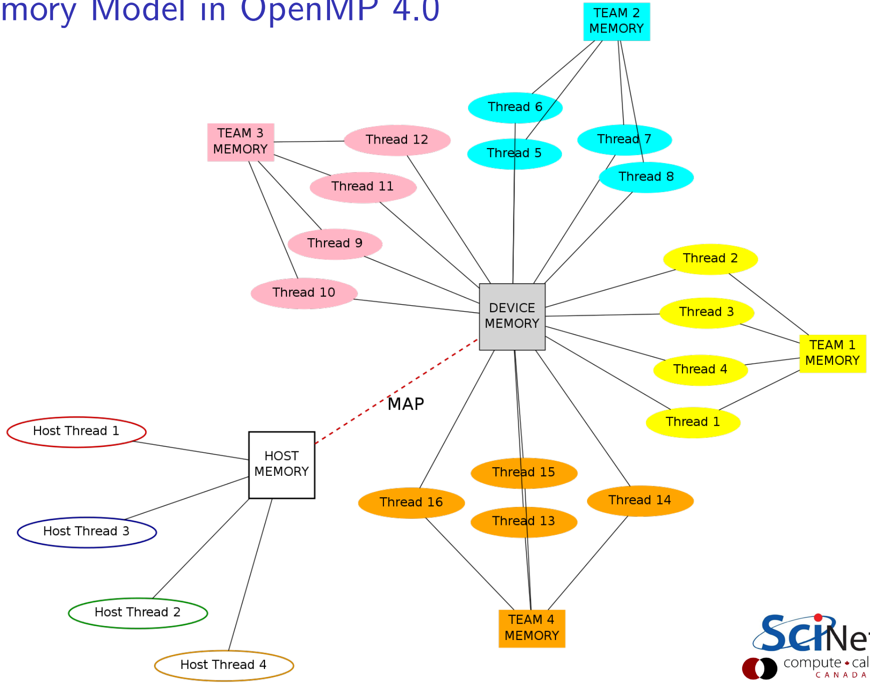
  GPUs, Xeon Phis, clusters(?)

- OpenMP 4.0 adds mechanisms to describe regions of code where data and/or computation should be moved to another computing device.

- Moves away from shared memory per se.

- `omp target`.

Memory Model in OpenMP 4.0



Memory Model in OpenMP 4.0

- Device has its own data environment
- And its own shared memory
- Threads can be bundled in a teams of threads
- These threads can have memory shared among threads of the same team
- Whether this is beneficial depends on the memory architecture of the device. (team $\approx$ CUDA thread blocks, MPI_COMM?)

Data mapping

- Host memory and device memory usually district.
- OpenMP 4.0 allows host and device memory to be shared.
- To accommodate both, the relation between variables on host and memory gets expressed as a *mapping*

  Different types:

  - `to`: existing host variables copied to a corresponding variable in the target before
  - `from`: target variables copied back to a corresponding variable in the host after
  - `tofrom`: Both `from` and `to`
  - `alloc`: Neither `from` nor `to`, but ensure the variable exists on the target but no relation to host variable.

  Note: arrays and array sections are supported.

OpenMP Device Example using target

```
/* example2.c */
#include <stdio.h>
#include <omp.h>
int main()
{
  int host_threads, trgt_threads;
  host_threads = omp_get_max_threads();
  #pragma omp target map(from:target_threads)
  trgt_threads = omp_get_max_threads();
  printf("host_threads = %d\n", host_threads);
  printf("trgt_threads = %d\n", trgt_threads);
}
```

```
> $CC -fopenmp example2.c -o example2
> ./example2
host_threads = 16
trgt_threads = 224
```

OpenMP Device Example using target

```
program example2
  use omp_lib
  integer host_threads, trgt_threads
  host_threads = omp_get_max_threads()
  !$omp target map(from:target_threads)
  trgt_threads = omp_get_max_threads();
  !$omp end target
  print *, "host_threads =", host_threads
  print *, "trgt_threads =", trgt_threads
end program example2
```

```
> $FC -fopenmp example2.f90 -o example2
> ./example2
     host_threads = 16
     trgt_threads = 224
```

OpenMP Device Example using teams, distribute

```
#include <stdio.h>
#include <omp.h>
int main()
{
   int ntprocs;
   #pragma omp target map(from:ntprocs)
   ntprocs = omp_get_num_procs();
   int ncases=2240, nteams=4, chunk=ntprocs*2;

   #pragma omp target
   #pragma omp teams num_teams(nteams) thread_limit(ntprocs/nteams)
   #pragma omp distribute
   for (int starti=0; starti<ncases; starti+=chunk)
      #pragma omp parallel for
      for (int i=starti; i<starti+chunk; i++)
         printf("case i=%d/%d by team=%d/%d thread=%d/%d\n",
                i+1, ncases,
                omp_get_team_num()+1, omp_get_num_teams(),
                omp_get_thread_num()+1, omp_get_num_threads());
}
```

OpenMP Device Example using teams, distribute

```
program example3
 use omp_lib
 integer i, ntprocs, ncases, nteams, chunk
 !$omp target map(from:ntprocs)
 ntprocs = omp_get_num_procs()
 !$omp end target
 ncases=2240
 nteams=4
 chunk=ntprocs*2
 !$omp target
 !$omp teams num_teams(nteams) thread_limit(ntprocs/nteams)
 !$omp distribute
 do starti=0,ncases,chunk
  !$omp parallel do
  do i=starti,starti+chunk
   print *,"i=",i,"team=",omp_get_team_num(),"thread=",omp_get_thread_num()
  end do
  !$omp end parallel
 end do
 !$omp end target
end program example3
```

Summary of directives

- omp target [map]
  marks a region to execute on device
- omp teams
  creates a league of thread teams
- omp distribute
  distributes a loop over the teams in the league
- omp declare target / omp end declare target marks function(s) that can be called on the device
  - map maps the computation onto a device and some number of threads on that device.
  - data allows the target to specify a region where data that is defined on the host is mapped onto the device, and sent (received) at the beginning (end) of the target region.

  ```
  #pragma omp target device(mic0) data map(to: v1[0:N], v2[:N]) map(from: p[0:N])
  ```
- omp get team num()
  omp get team size()
  omp get num devices()

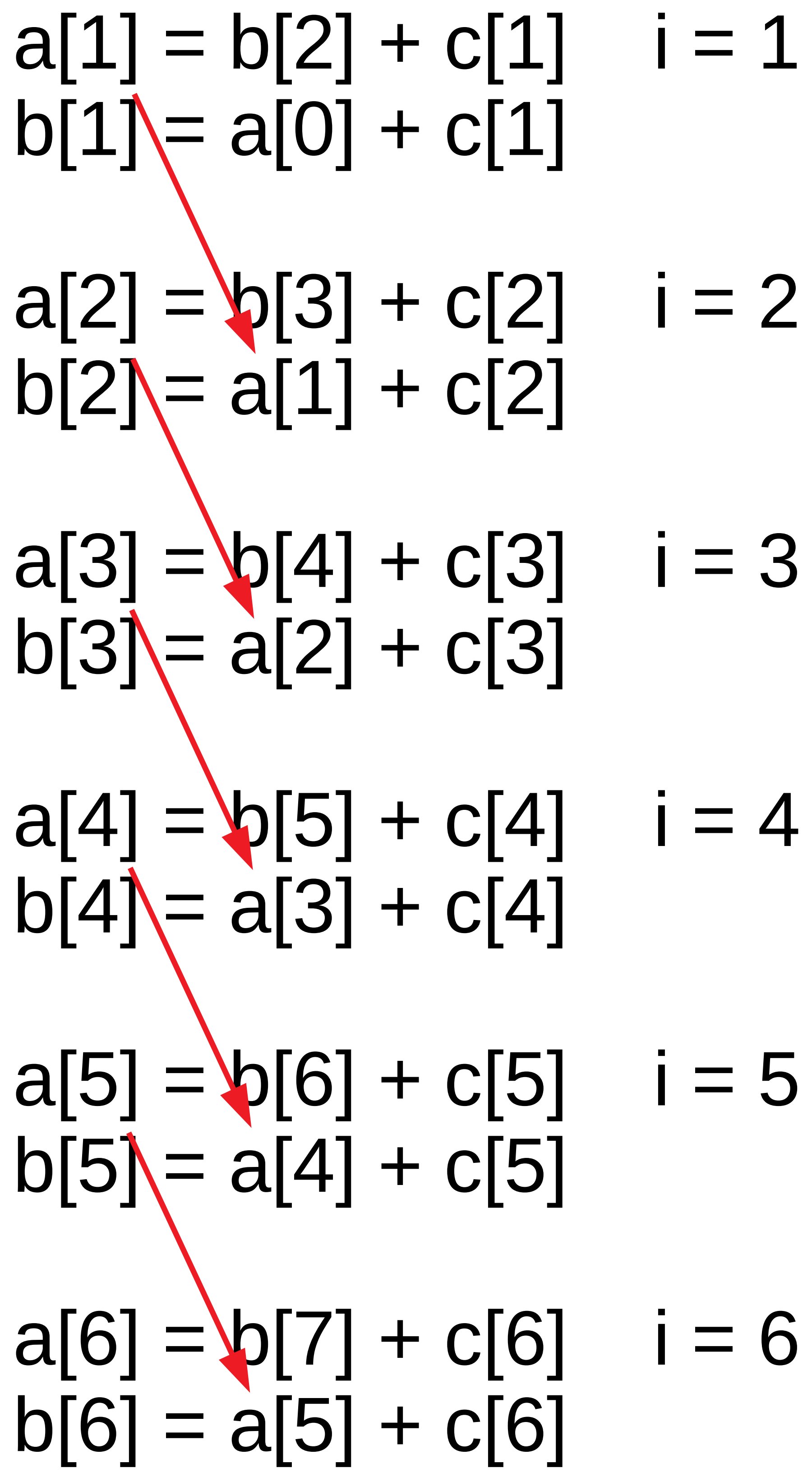
Vector parallelism (SIMD parallelization)

Consider the loop

```
for (int i = 1; i < n; i++) {
   a[i] = b[i+1] + c[i]

   b[i] = a[i-1] + c[i]
}
```

Because of the dependence on a, we cannot execute this as a single parallel loop in OpenMP. We can execute it as two parallel loops, i.e.,

```
#pragma omp parallel for
for (int i = 1; i < n; i++) {
   a[i] = b[i+1] + c[i]
}
#pragma omp parallel for
   b[i] = a[i-1] + c[i]
}
```



```
a[1] = b[2] + c[1]    i = 1
b[1] = a[0] + c[1]

a[2] = b[3] + c[2]    i = 2
b[2] = a[1] + c[2]

a[3] = b[4] + c[3]    i = 3
b[3] = a[2] + c[3]

a[4] = b[5] + c[4]    i = 4
b[4] = a[3] + c[4]

a[5] = b[6] + c[5]    i = 5
b[5] = a[4] + c[5]

a[6] = b[7] + c[6]    i = 6
b[6] = a[5] + c[6]
```

What are other ways of exploiting the latent parallelism in this loop?

Dataflow is one.

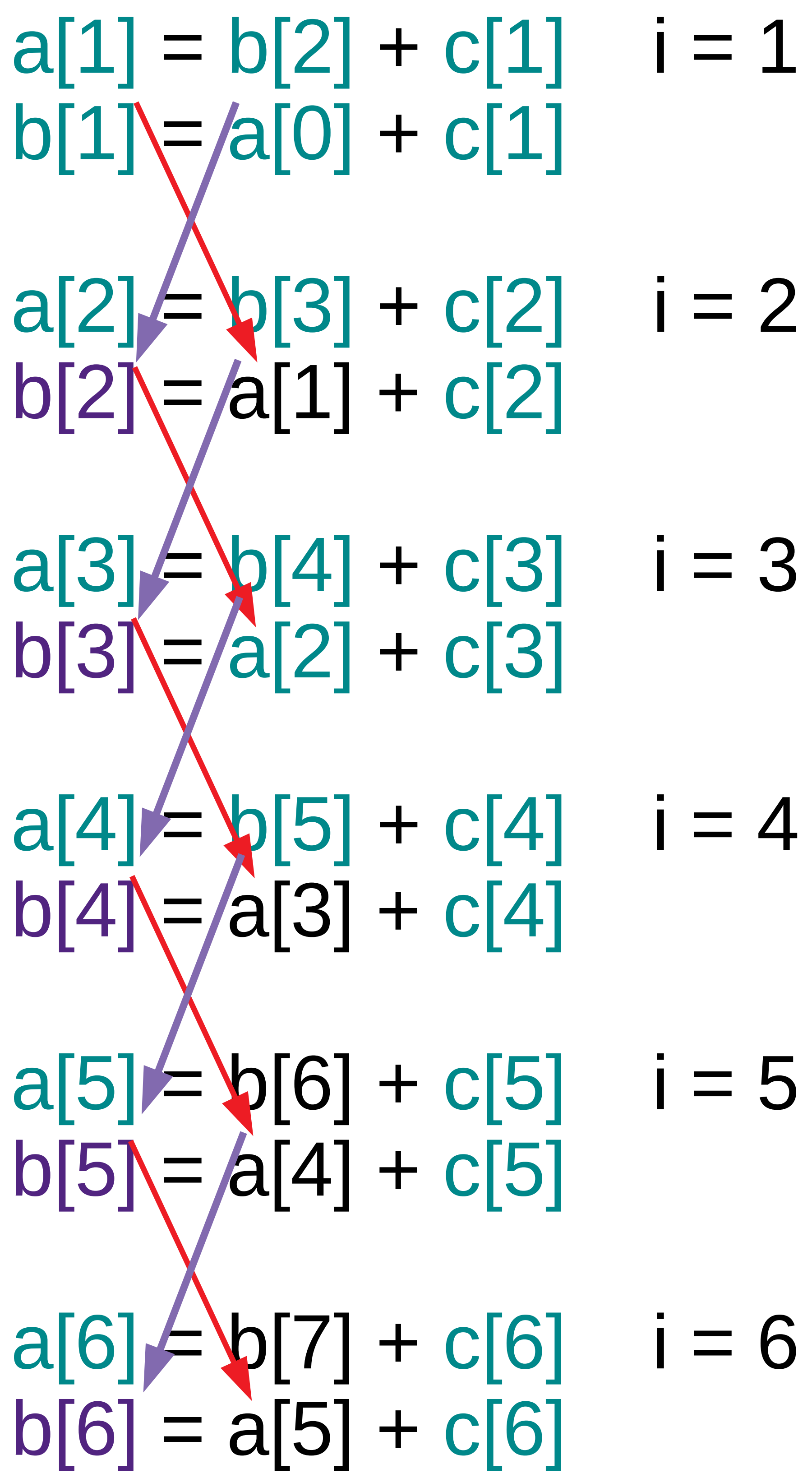
Dataflow

As soon as the operands for an operation are ready, perform the operation.

Green operands are operands that are ready at step 1.

Red operands are operands that must wait for a value to be produced. (*true* or *flow dependence* in compiler terminology.

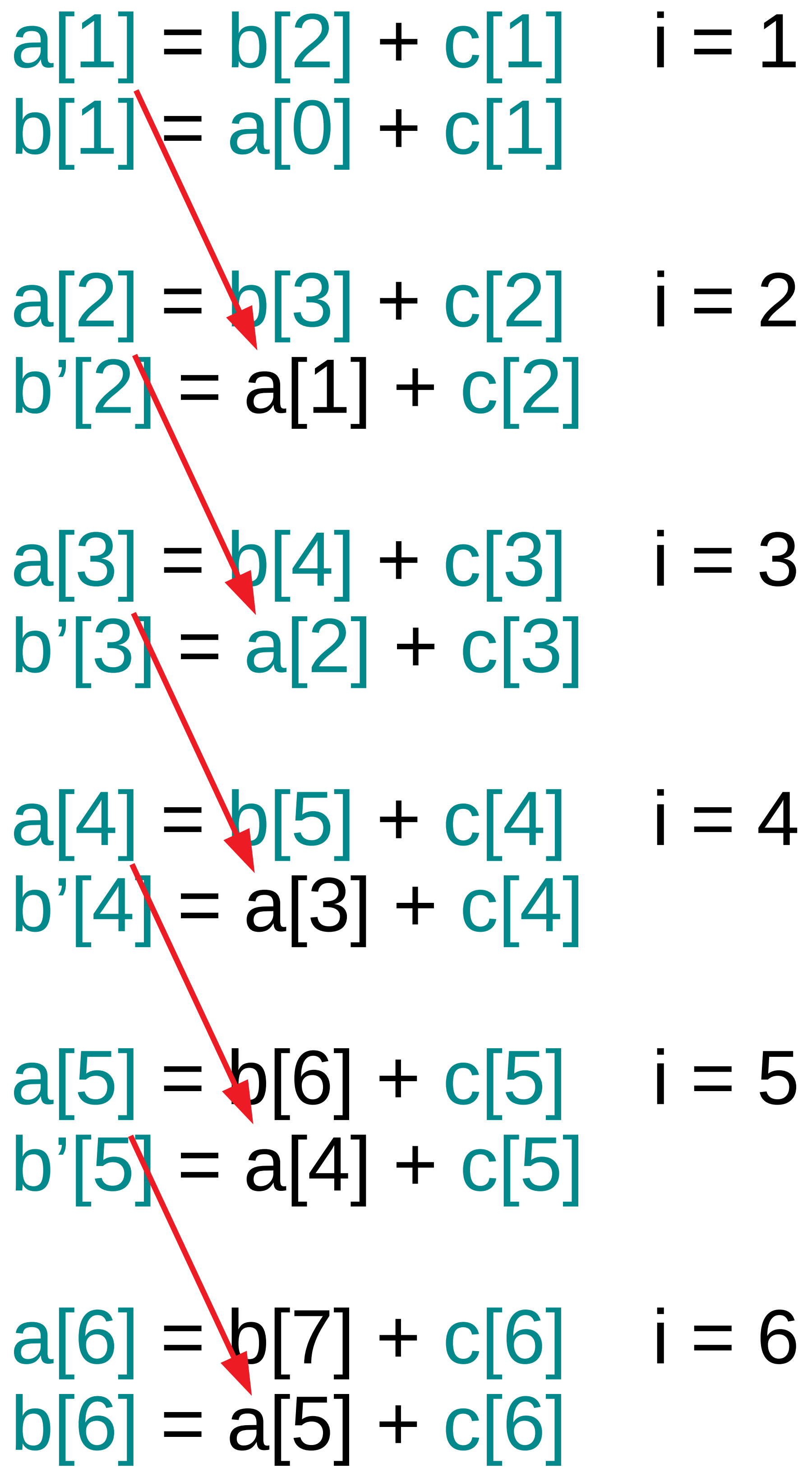
Purple operands are operands that must wait for a value to be produced. (*anti dependence* in compiler terminology



```
a[1] = b[2] + c[1]    i = 1
b[1] = a[0] + c[1]

a[2] = b[3] + c[2]    i = 2
b[2] = a[1] + c[2]

a[3] = b[4] + c[3]    i = 3
b[3] = a[2] + c[3]

a[4] = b[5] + c[4]    i = 4
b[4] = a[3] + c[4]

a[5] = b[6] + c[5]    i = 5
b[5] = a[4] + c[5]

a[6] = b[7] + c[6]    i = 6
b[6] = a[5] + c[6]
```

Anti dependences can be eliminated with extra storage

Create alternate b elements.
We won’t worry about how to address these.



```
a[1] = b[2] + c[1]     i = 1
b[1] = a[0] + c[1]

a[2] = b[3] + c[2]     i = 2
b’[2] = a[1] + c[2]

a[3] = b[4] + c[3]     i = 3
b’[3] = a[2] + c[3]

a[4] = b[5] + c[4]     i = 4
b’[4] = a[3] + c[4]

a[5] = b[6] + c[5]     i = 5
b’[5] = a[4] + c[5]

a[6] = b[7] + c[6]     i = 6
b[6] = a[5] + c[6]
```

All statements can be executed in 2 steps given sufficient hardware

T=1
a[1] = b[2] + c[1], a[2] = b[3] + c[2],
a[3] = b[4] + c[3], a[4] = b[5] + c[4],
a[5] = b[6] + c[5], a[6] = b[7] + c[6]

~~a[1] = b[2] + c[1]~~ i = 1
b[1] = a[0] + c[1]

~~a[2] = b[3] + c[2]~~ i = 2
b’[2] = a[1] + c[2]

~~a[3] = b[4] + c[3]~~ i = 3
b’[3] = a[2] + c[3]

~~a[4] = b[5] + c[4]~~ i = 4
b’[4] = a[3] + c[4]

~~a[5] = b[6] + c[5]~~ i = 5
b’[5] = a[4] + c[5]

~~a[6] = b[7] + c[6]~~ i = 6
b[6] = a[5] + c[6]

All statements can be executed in 2 steps given sufficient hardware

T=1
a[1] = b[2] + c[1], a[2] = b[3] + c[2],
a[3] = b[4] + c[3], a[4] = b[5] + c[4],
a[5] = b[6] + c[5], a[6] = b[7] + c[6]

T=2
b[1] = a[0] + c[1],
b'[2] = a[1] + c[2], b'[3] = a[2] + c[3],
b'[4] = a[3] + c[4], b'[5] = a[4] + c[5],
b[6] = a[5] + c[6]

We are done in two time steps!

~~a[1] = b[2] + c[1]~~ i = 1
b[1] = a[0] + c[1]

~~a[2] = b[3] + c[2]~~ i = 2
b'[2] = a[1] + c[2]

~~a[3] = b[4] + c[3]~~ i = 3
b'[3] = a[2] + c[3]

~~a[4] = b[5] + c[4]~~ i = 4
b'[4] = a[3] + c[4]

~~a[5] = b[6] + c[5]~~ i = 5
b'[5] = a[4] + c[5]

~~a[6] = b[7] + c[6]~~ i = 6
b[6] = a[5] + c[6]

MIT Monsoon project was the largest data flow machine created

- Storage freeing was an issue, array layout was another one
- Ran out of storage in a couple of weeks. Could a different language and garbage collection help? Probably not for array-based numerical languages.
- Auto-parallelization failed because false unnecessary dependences prevent parallelization
- Data-flow failed because of hardware complexity and the inability of data dependence to precisely show when storage could be freed.
- Note that *register renaming* and *array privatization* are two techniques that break anti-dependences and allow better auto-parallelization. One of the most important techniques.

Dataflow is not dead, however

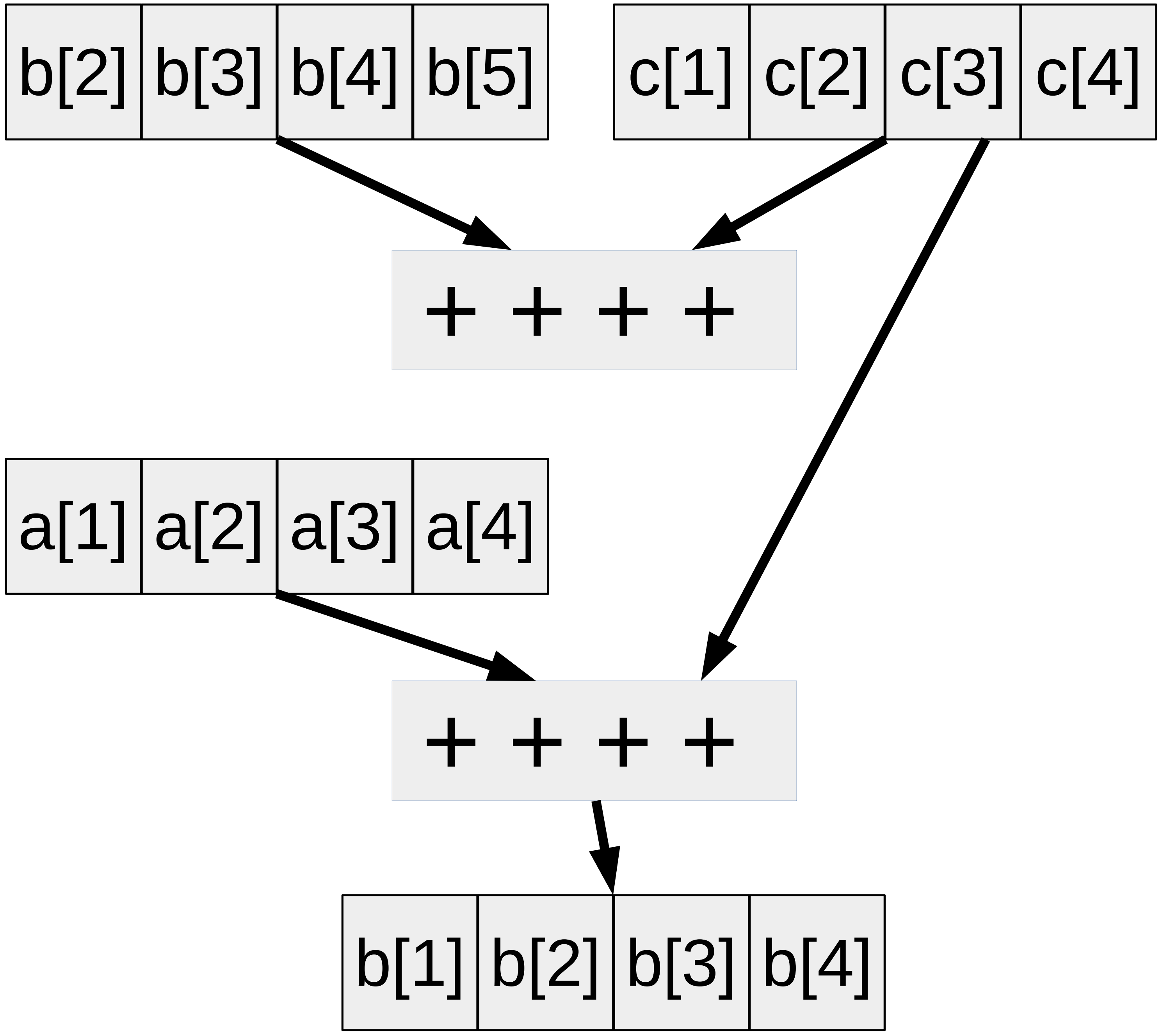
- Most modern processors implement Tomasulo’s algorithm, or a variation of it.
- First used in the IBM 360/91, 16.6M instructions/sec
- Enables out-of-order instruction execution to use multiple functional units in a processor
- Parallelism for free, at least in terms of programmer time.

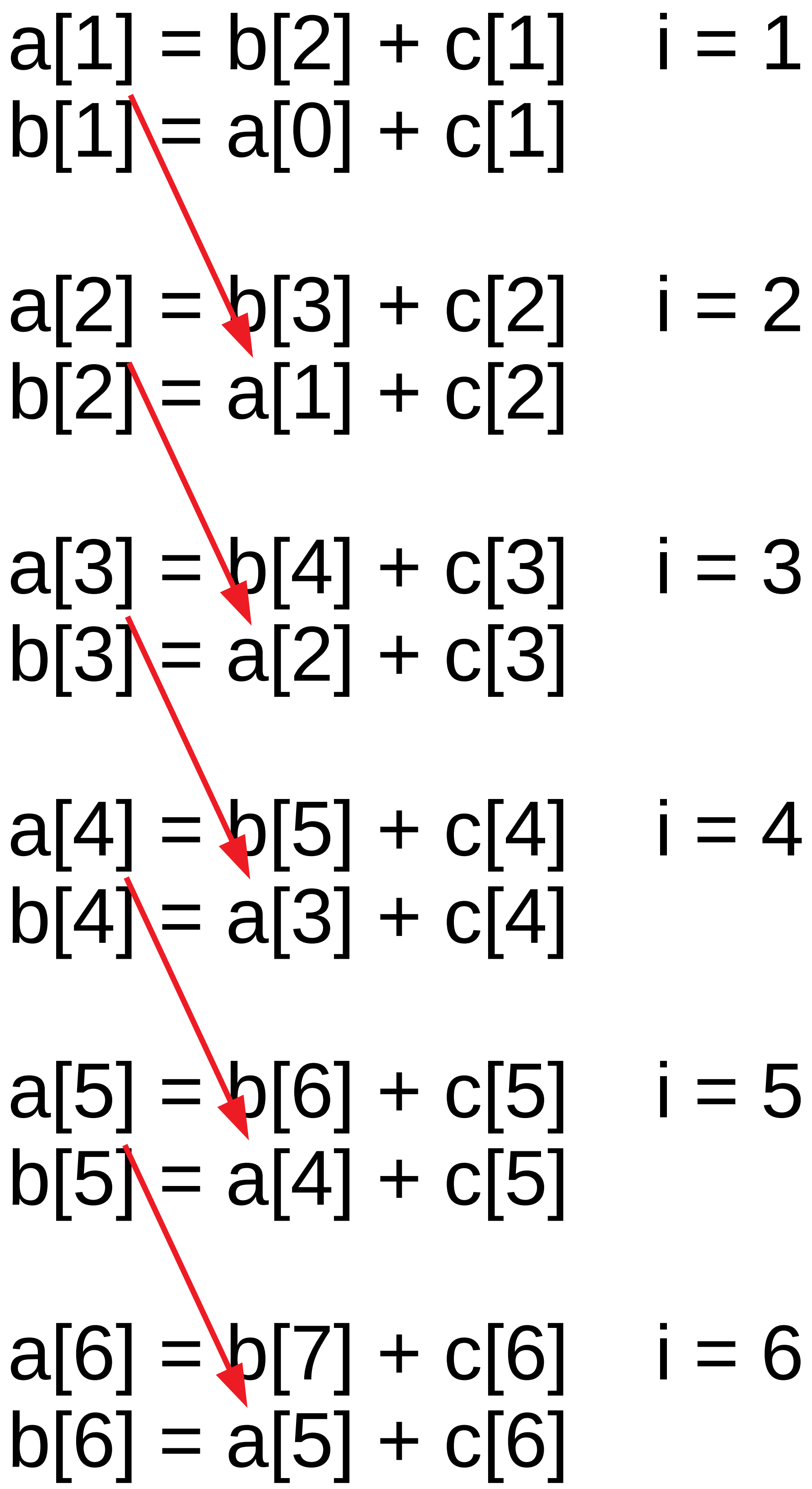
Problems with this approach

- This is not under programmer control — the programmer only specifies the instructions to be executed, not the functional unit that executes the instruction.
- Normal multi-functional unit processors need circuitry to control and fire the functional units (Tomasulo’s algorithm)
- Hardware must detect availability of operands and functional unit, and schedule the operation onto a particular hardware functional unit. Enables *out-of-order* execution.
- Vectors allow multiple operations to occur with less control logic.

Vector execution

```
for (int i = 1; i < n; i++) {
  a[i] = b[i+1] + c[i]

  b[i] = a[i-1] + c[i]
}
```





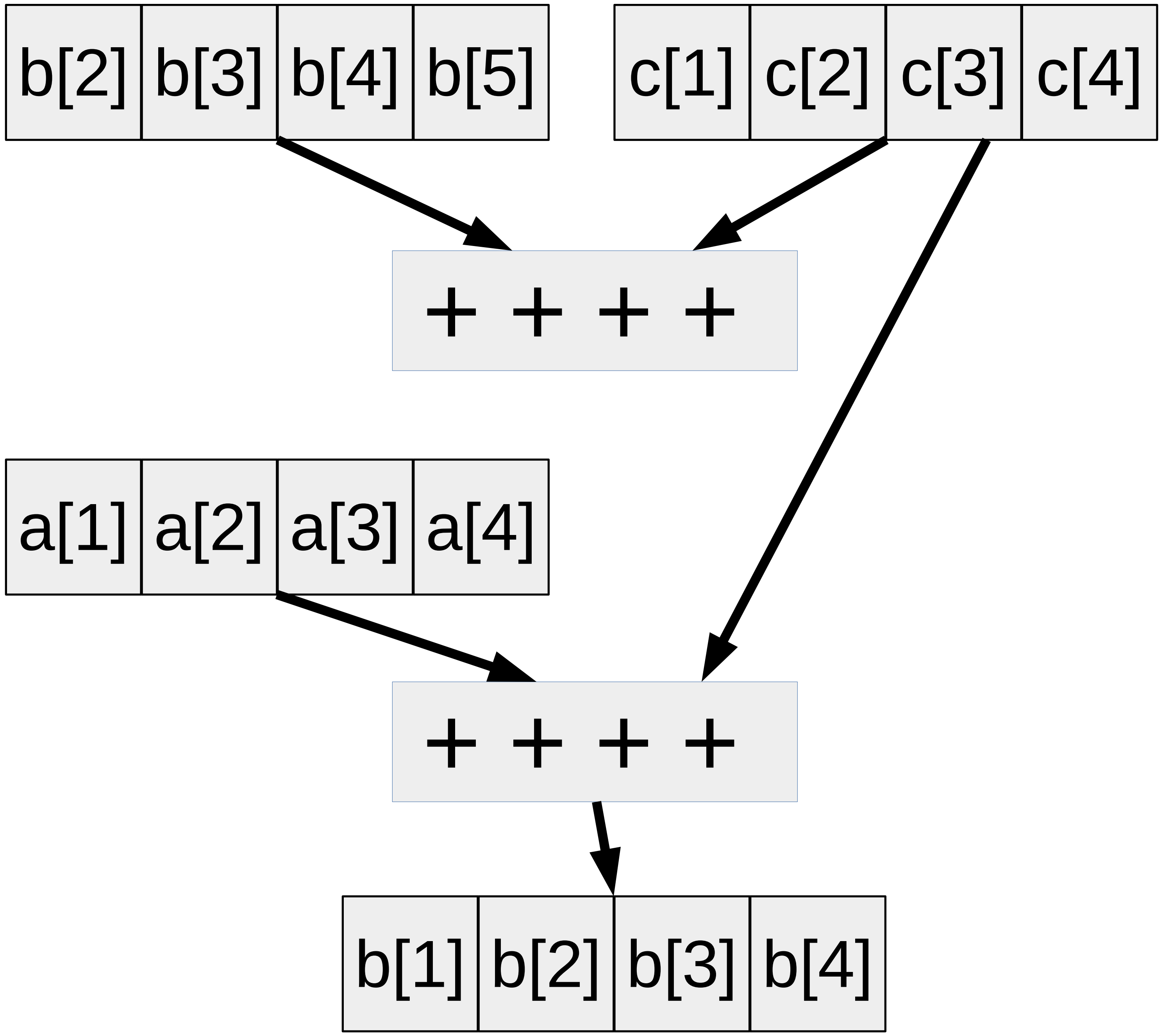
```
a[1] = b[2] + c[1]    i = 1
b[1] = a[0] + c[1]

a[2] = b[3] + c[2]    i = 2
b[2] = a[1] + c[2]

a[3] = b[4] + c[3]    i = 3
b[3] = a[2] + c[3]

a[4] = b[5] + c[4]    i = 4
b[4] = a[3] + c[4]

a[5] = b[6] + c[5]    i = 5
b[5] = a[4] + c[5]

a[6] = b[7] + c[6]    i = 6
b[6] = a[5] + c[6]
```

Why vectors are good

- With vector units there are architected vector registers and vector functional units
- These work on groups, or vectors of operands and operations
- Programmer/compiler generates the instructions
- Control in hardware is almost no more complicated than a scalar functional unit
- Allows more operations to be done per clock with small increase in processor complexity

Vector execution

```
for (int i = 1; i < n; i++) {
  a[i] = b[i+1] + c[i]

  b[i] = a[i-1] + c[i]
}
```

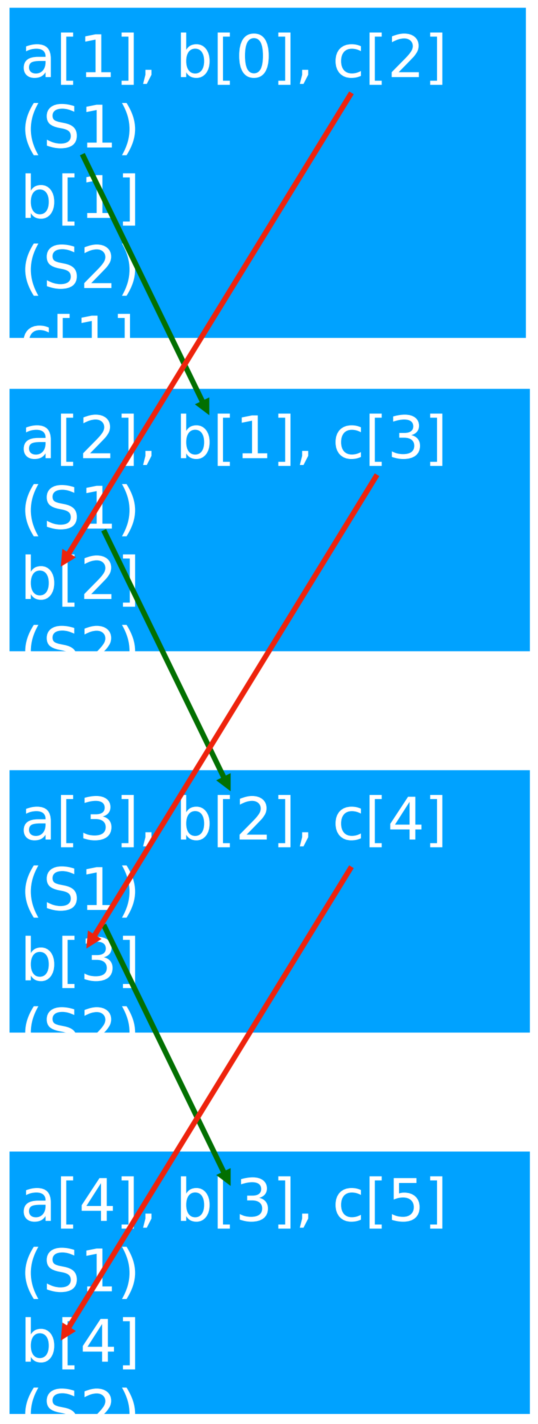


4 operations in a time step
No complicated control circuitry needed

Modern Intel processors have 2 512 AVX units, allowing them to execute 32 DP ops / cycle, and up to 2 512 DP FMA / cycle

Vector parallelization

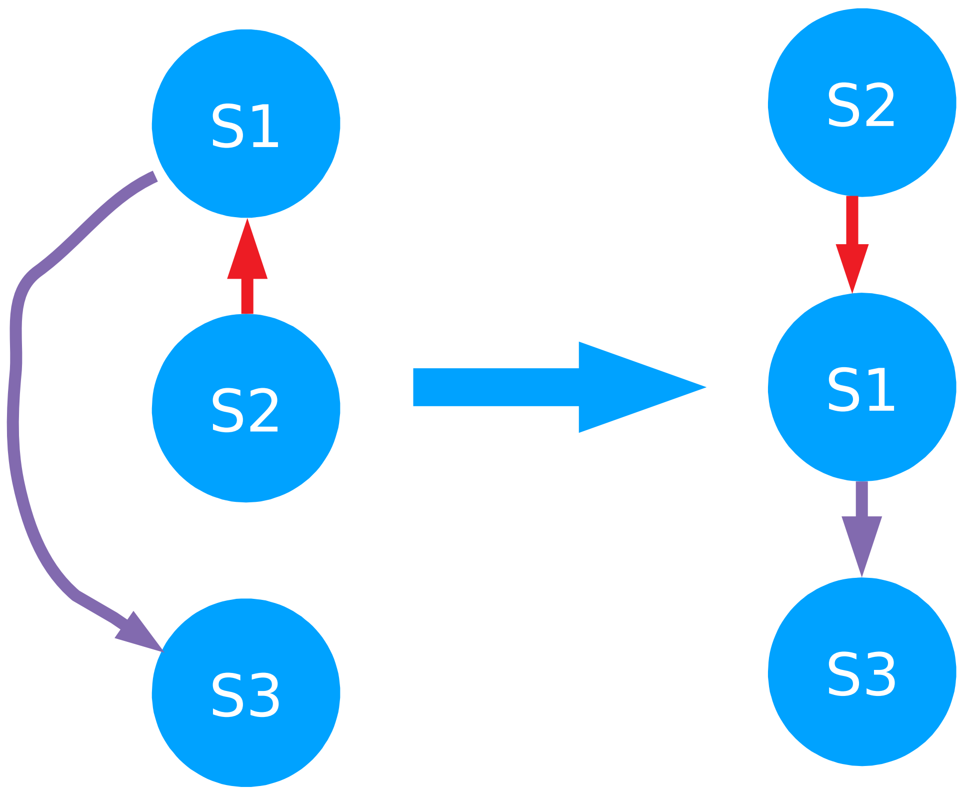
```
for (int i = 1, i < n, i++) {

   a[i] = b[i-1] + c[i+1]; (S1)

   b[i] = d[i] + e[i]; (S2)

   c[i] = f[i] + g[i]; (S3)

}
```



Dependences go from earlier to later statements. This is not good, as executing 4 iterations of S1 before S2 will cause S1 to get stale values.

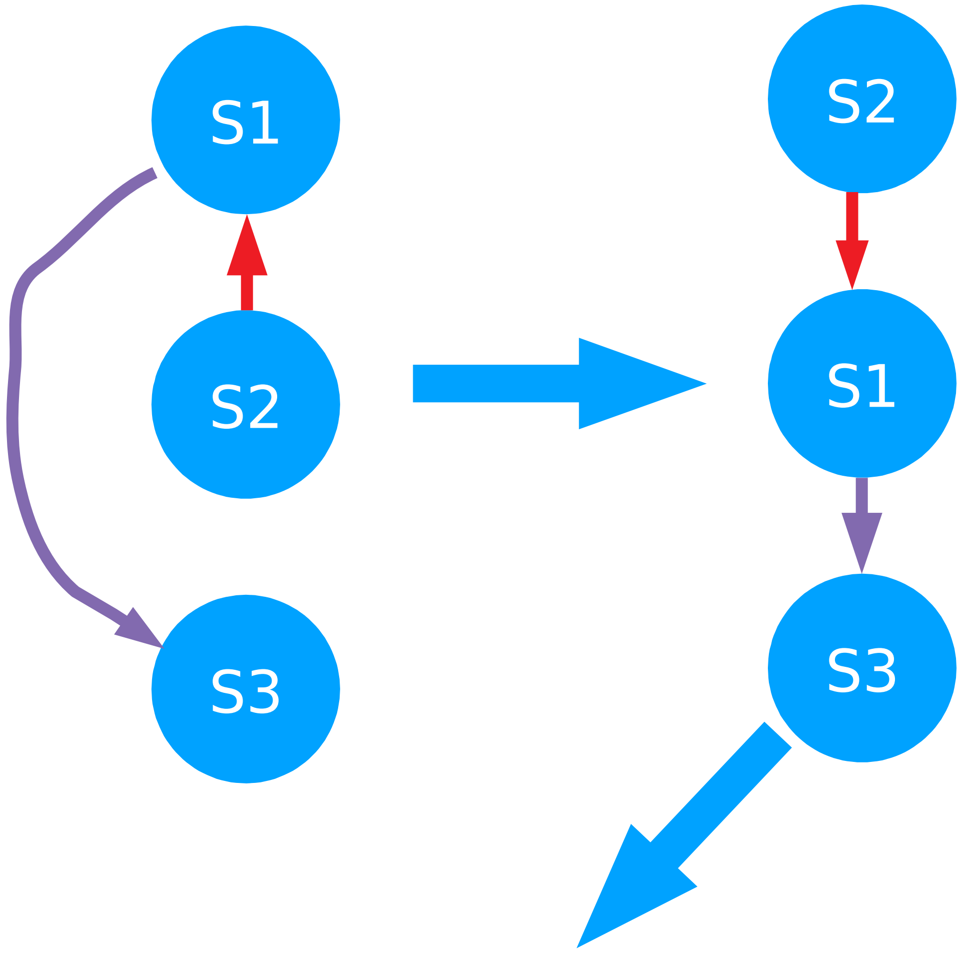
Vector parallelization

```
for (int i = 1, i < n, i++) {

   a[i] = b[i-1] + c[i+1]; (S1)

   b[i] = d[i] + e[i]; (S2)

   c[i] = f[i] + g[i]; (S3)

}
```



Vector parallelization

```
for (int i = 1, i < n, i++) {

   a[i] = b[i-1] + c[i+1]; (S1)

   b[i] = d[i] + e[i]; (S2)

   c[i] = f[i] + g[i]; (S3)

}
```



```
for (int i = 1, i < n, i++) {
    b[i] = d[i] + e[i]; (S2)

    a[i] = b[i-1] + c[i+1]; (S1)

    c[i] = f[i] + g[i]; (S3)
}
```

Vector parallelization

```
for (int i = 1, i < n, i++) {
   b[i] = d[i] + e[i]; (S2)

   a[i] = b[i-1] + c[i+1]; (S1)

   c[i] = f[i] + g[i]; (S3)
}
```

```
for (int i = 1, i < n, i++) {
   b[i] = d[i] + e[i]; (S2)
}

for (int i = 1, i < n, i++) {
   a[i] = b[i-1] + c[i+1]; (S1)
}

for (int i = 1, i < n, i++) {
   c[i] = f[i] + g[i]; (S3)
}
```

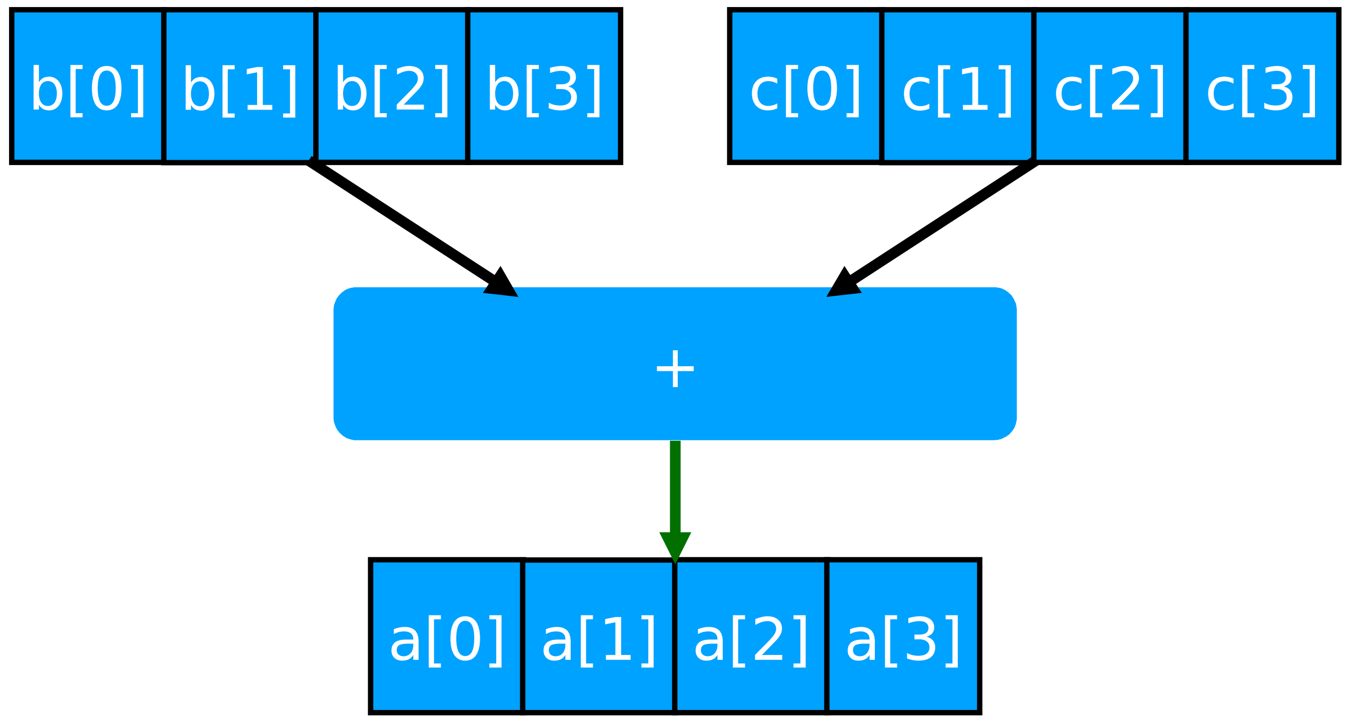
```
for (int i = 1, i < n, i+=4) {
   vadd b[i], d[i], e[i]; (S2)
}

for (int i = 1, i < n, i+=4) {
   vadd a[i], b[i-1], c[i+1]; (S1)
}

for (int i = 1, i < n, i+=4) {
   vadd c[i], f[i], g[i]; (S3)
}
```

Why vectors?

```
For (int i = 0; i < n;
      i++) {
   a[i] = b[i]*c[i];
}
```

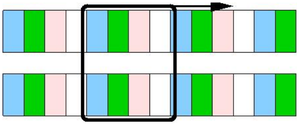
```
For (int i = 0; i < n; i+=4) {
   a[i] = b[i]*c[i];
   a[i+1] =     b[i+1]*c[i+1];
   a[i] = b[i+2]*c[i+2];
   a[i] = b[i+2]*c[i+2];
}
```

```
For (int i = 0; i < n; i+=4)
{
   ldv rv1, b[i]
   ldv rv2, c[i]
   vadd rv3, rv1, rv2
}
```



- With vector units there are architected vector registers and vector functional units
- These work on groups, or vectors of operands and operations
- Programmer/compiler generates the instructions
- Control in hardware is almost no more complicated than a scalar functional unit
- Allows more operations to be done per clock with small increase in processor complexity

2. SIMD Constructs



- OpenMP can enable vectorization of both serial as well as parallelized loops.
- *vectorization* = processing multiple elements of an array at the same time.
- This is done using SIMD instructions.
- SIMD=single instruction multiple data. Usually 2, 4,or 8 *SIMD lanes* wide.
- Can also indicate to OpenMP to create versions of functions that can be invoked across SIMD lanes.

New Directives for SIMD Support

- `omp simd`

  marks a loop to be executed using SIMD lanes

- `omp declare simd`

  marks a function that can be called from a SIMD loop

- `omp parallel for simd`

  marks a loop for thread work-sharing as well as SIMDing

OpenMP SIMD Loop Example

```
#include <stdio.h>
#define N 262144
int main()
{
   long long d1=0;
   double a[N], b[N], c[N], d2=0.0;
   #pragma omp simd reduction(+:d1)
   for (int i=0;i<N;i++)
      d1+=i*(N+1-i);
   #pragma omp simd
   for (int i=0; i<N;i++) {
      a[i]=i;
      b[i]=N+1-i;
   }
   #pragma omp parallel for simd reduction(+:d2)
   for (int i=0; i<N; i++)
      d2+=a[i]*b[i];
   printf("result1 = %ld\nresult2 = %.2lf\n", d1, d2);
}
```

OpenMP SIMD Loop Example

```
program simdex
  integer, parameter :: N = 262144
  integer(kind=8) :: i, d1
  real(kind=8), dimension(N) :: a, b, c
  real(kind=8) :: d2
  d1=0 ; d2=0.
  !$omp simd reduction(+:d1)
  do i=1,N
     d1 = d1 + (i-1)*(N-i)
  end do
  !$omp end simd
  !$omp simd
  do i=1,N
     a(i)=i-1 ; b(i)=N-i
  end do
  !$omp end simd
  !$omp parallel do simd reduction(+:d2)
  do i=1,N
     d2 = d2 + a(i)*b(i)
  enddo
  !$omp end parallel
  print *,"result1 =",d1,"result2 =",d2
end program simdex
```

OpenMP SIMD Function Example

```
#include <stdio.h>
#pragma omp declare simd
double computeb(int i)
{ return N+1-i; }
#define N 262144
int main()
{
  long long d1=0;
  double a[N], b[N], c[N], d2=0.0;
  #pragma omp simd reduction(+:d1)
  for (int i=0;i<N;i++)
    d1 += i*computeb(i);
  #pragma omp simd
  for (int i=0; i<N;i++) {
    a[i]=i; b[i]=computeb(i);
  }
  #pragma omp parallel for simd reduction(+:d2)
  for (int i=0; i<N; i++)
    d2 += a[i]*b[i];
  printf("result1 = %ld\nresult2 = %.2lf\n", d1, d2);
}
```

3. Task Enhancements

- Can abort parallel OpenMP execution by conditional cancellation at implicit and user-defined cancellation points.
- Tasks can be grouped to into task groups can be aborted to reflect completion of cooperative tasking activities such as search.
- Task-to-task synchronization is supported through the specification of task dependency.

OpenMP Task Cancellation Example

```
#include <stdio.h>
#define N 40
int main()
{
   char haystack[N+1]="abcabcabczabcabcabcxabcabcabczabcabcabcz";
   char needle='x';
   int pos;
   #pragma omp parallel for
   for (int i=0; i<N; i++) {
      if (haystack[i]==needle) {
         pos=i;
         #ifndef _OPENMP
         break;
         #else
         #pragma omp cancel for
         #endif
      }
   }
   printf("\n'%c' found at position %d in %s\n",needle,pos,haystack);
}
```

Overview of New Directives and Functions for Tasks

- `omp cancel parallel|for|sections|taskgroup`
  starts cancellation of all tasks in the same construct
- `omp cancelation point parallel|for|sections|taskgroup`
  marks a point at which this task may be canceled
- `omp taskgroup`
  marks a region such that all tasks started in it belong to a group
- `omp task depend([in|out|inout]:`*`variable`*`)` clause
  marks that a task depends on other task

Concurrent Execution w/ Dep.

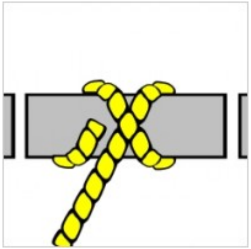
- **Note: variables in the `depend` clause do not necessarily have to indicate the data flow**

T1 has to be completed before T2 and T3 can be executed.

T2 and T3 can be executed in parallel.

```
void process_in_parallel) {
   #pragma omp parallel
   #pragma omp single
   {
      int x = 1;
      ...
      for (int i = 0; i < T; ++i) {
         #pragma omp task shared(x, ...) depend(out: x)  // T1
            preprocess_some_data(...);
         #pragma omp task shared(x, ...) depend(in: x)   // T2
            do_something_with_data(...);
         #pragma omp task shared(x, ...) depend(in: x)   // T3
            do_something_independent_with_data(...);
      }
   } // end omp single, omp parallel
}
```

4. Thread Affinity



- OpenMP can now be told better where to execute threads.
- Can be used to get better locality, less false sharing, more memory bandwidth.
- To specify platform-specific data: Environment variable `OMP_PLACES`
- To describe thread binding to processor:
  - Environment variable: `OMP_PROC_BIND`
  - In code using `omp parallel`'s new `proc_bind` clause.

Allowed values:
`false, true, master, close, spread`

Example of Specifying Affinity See http://pages.tacc.utexas.edu/~eijkhout/pcse/html/omp-affinity.html for more examples.

OMP_PLACES=sockets
On a node with two processors (i.e., two sockets, each of which contains a processor) and each processor has 8 cores, this will place threads like:
Processor0 (socket 0) = {t0, t2, t4, t6, t8, t10, t12, t14}
Processor1 (socket 1) = {t1, t3, t5, t7, t9, t11, t13, t15}

In the same system, if you specify
OMP_PLACES=cores
OMP_PROC_BIND=close
You will get
Processor0 (socket 0), cores 0 - 7 have threads = t0, t1, t2, t3, t4, t5, t6, t7
Processor1 (socket 1), cores 8 - 15 = t8, t9, t10, t11, t12, t13, t14, t15

On the same system, if you specify
OMP_PLACES=cores
OMP_PROC_BIND=close
You will get
T0 == core0, t1 == core 8, t2 = core1, t3 = core9, t4 = core2, t5 = core10, …, t14 = core7, thread15 = core15
Processor1 (socket 1) = {t1, t3, t5, t7, t9, t11, t13, t15}

This is similar to OMP_PLACES=sockets, except that OMP_PLACES=sockets does not bind a thread to a particular core, only to a particular socket

You can also specify
OMP_PLACES=0,8,1,9,2,10,3,11,4,12,5,13,6,14,7,15
And this will cause placing work to treat 0 and 8 as close, 8 and 1 as close, etc.

5. Other improvements

- User-defined reductions:
  Previously, OpenMP API only supported reductions with base language operators and intrinsic procedures. With OpenMP 4.0 API, user-defined reductions are now also supported.

  ```
  omp declare reduction
  ```

- Sequentially consistent atomics:
  A clause has been added to allow a programmer to enforce sequential consistency when a specific storage location is accessed atomically.

  ```
  omp atomic seq_cst
  ```

- Optional dump all internal variables at program start:

  ```
  OMP_DISPLAY_ENV=TRUE|FALSE|VERBOSE
  ```