Chapter 2: designing Parallel Algorithms

A four-part strategy:

- Partition
- Identify communication links
- Agglomerate
- Map onto computer

Not necessarily to be performed linearly or in this order.

2.1 Partitioning

- Expose potential parallelism -- finegrained decomposistion fo the problem
- Divide both computation and data:

  1. Domain decomposistion: partition data, associate operations (data parallel)

  2. Functional decomposition: partition computations that need to be performed, and identify data transfer (MIMD).
- A natural approach for some problems: functional decomposition first and then domain decomposition.

2.1 Partitioning (Cont.)

A very simple illustration of the two partitioning strategies is presented by the two parallel loop techniques discussed earlier:

Consider the loop:

```
do i=1,n
   a(i)=b(i)+1
   c(i)=c(i-1)+a(i)
   d(i)=c(i)+e(i)
end do
```

Functional decomposition (dopipe):

```
cobegin
   do i=1,n
      a(i)=b(i)+1
      post e(i)
   end do
//
   do i=1,n
      wait e(i)
      c(i) =  c(i-1)+a(i)
end do
...
```

Domain decomposition:

1. Doacross

```
doacross i=1,n
   a(i)=..
   wait e(i-1)! alterntively c$omp ordered...
      c(i)=c(i-1) ...
   post e(i)   ! alternatively c$omp end oredered
   ...
```

2. Doall

```
doall i=1,n
   a(i)=..
end do
call parallel_linear_recurrence_solver(c,a)
doall i=1,n
   d(i) = ...
end do
```

2.2 Communication.

Once the problem has been partitioned, we need to investigate the communication needs between subparts. Communication is critical in the design of parallel algoritms.

- Try to minimize communication (trade off with parallelism)
- Local vs. Global communication
- Structured vs. Unstructured communication
- Static vs. Dynamic communication
- Synchronous vs. Asynchronous communication.

2.2 Communication (Cont.)

In shared memory, communication is implicit.

However, it is usually important to be aware of communication costs.

However, determining costs could be quite complex depending on the cache coherency protocol.

Some suggest that parallel shared-memory programs should be designed as message-passing programs and then translated into shared-memory form (by replacing send/ receives with assignments).

2.3 Agglomeration

Once the problem has been partitioned, and the communication need determined, it sometimes help to combine tasks in order to reduce communication requirements.

- Costs associated with message size, message creation, synchronization operations, and task creation.
- Agglomeration increases granularity and thus reduces communication.
- However, it is advisable to have high granularity in order to maintain flexibility.

2.4 Mapping

- The general problem is NP-complete

- Rules of Thumb:

1. Tasks that can execute concurrently mapped to different processors.
2. Tasks that communicate a lot on same processor

- Some domain decomposition tasks result in structured communication and a straightforward mapping.

- More complex problems require load balancing, and the most complex problems require dynamic load balancing.

Scheduling options in OpenMP

```
C$OMP PARALLEL DO ... SCHEDULE(STATIC,<chunk_size>)
C$OMP PARALLEL DO ... SCHEDULE(DYNAMIC,<chunk_size>)
C$OMP PARALLEL DO ... SCHEDULE(GUIDED,<chunk_size>)
```

- Static scheduling with chunk size of *n*. *n* iterations are dispatched statically to each thread (repeat until all iterations have been dispatched). If n is missing, it is assumed *I/p* where *I* is the total number of iterations and *p* the number of threads.
- Dynamic scheduling with chunk size of *n*. *n* iterations are dispatched dynamically to each thread. If n is missing, it is assumed 1.
- In Guided Self-Scheduling, each processor is dynamically allocated $\left\lceil \frac{R_i}{P} \right\rceil$ loop iterations where R_i is the number of iteration remaining at time t_i.

- The simplest scheduling algorithm is self-scheduling (`DYNAMIC` with chunk *1*)and consist of each processor fetching an iteration of the loop one at a time by (atomically) incrementing a shared conunter variable.

  Self scheduling performs the best possible load balancing because all processors are guaranteed to finish within the execution period of the loop iteration that finishes last.

- Block-scheduling (`DYNAMIC` with chunk *n>1*) reduces the overhead by having processors scheudle blocks of iterations at once. But load balancing is only accomplished to iwthin the execution time of the chunk of loop iterations to be completed last.

- In `GUIDED`, scheduling large chunks of iterations in the beginning results in low scheduling overhead, while scheduling small chunks towards the end provides good load balancing.