Contents

1 Dependencies

A **dependence** is an ordering relationship between two computations. For correct results the **ordering** must be observed. E.g.:

- One process (or thread) waiting for data from another process (or thread)
- Two threads/processes reading/writing the same memory locations

1.1 Flow dependence

A *Flow* dependency, also known as a *data* dependency or **true** dependency or *read-after-write* (RAW), occurs when an instruction depends on the result of a previous instruction:

$$A = 3; B = A; C = B;$$

In here last instruction is dependent on second one and second one is dependent on the first.

1.2 Anti dependence

Write after read (WAR): a **false** dependence, only caused by **reusing the memory location**. Occurs when an instruction requires a value that is later updated. May be able to eliminate it by using **separate memory**, e.g. different variables.

$$B = 3; A = B + 1; B = 7$$

This prevents the instructions from being executed in parallel

1.3 Output dependence

Write after write (WAW): Also a **false dependence** caused by **reusing** the memory location. May be able to eliminate it by using separate memory, e.g. different variables

$$B = 3; A = B + 1; B = 7$$

1.4 Input dependence

Read after Read (RAR): **Does NOT impose** an order constraint. So usually we dont need to worry about it at all, but it may tell us something about locality which might be useful for analysis

1.5 Dependencies and Parallelism

Dependencies (both true and false) **limit parallelism**. E.g. adding numbers in an array : lines are true dependencies, where one function needs the result from another before it can be evaluated. These dependencies and hence ordering constraints are inherent to the code. **Key point**: avoid introducing dependencies that **do not matter to the computation** they will probably limit parallelism.

2 Granularity

The granularity of a parallel computation is how much work (how many instructions) can be done **within a single thread or process** between each interaction with another thread or process. Or equivalently how infrequent **interaction** with other threads or processes

2.1 Fine-grained Parallelism

Fine (small) grained parallelism has **few instructions between interactions** interaction is **frequent**. E.g. the code generated by a parallelising compiler from a sequential program. Each thread is likely to make **frequent access to shared variables**, perhaps doing only a few instructions between each access. OK on a multi-core machine with hardware shared memory because **interaction is relatively fast**

2.2 Course-grained Parallelism

Coarse (large) grained parallelism **has many instructions between interactions** interaction is **infrequent**.
E.g. SETI@home, using home PCs connected to the Internet to analyse radio telescope data for evidence of extra-terrestrial intelligence. Each PC downloads enough **work for several hours or days**, uploaded **only when complete**. Very course-grained, because interaction (over the internet) is slow; Also makes use of the donors internet connection more efficient and time-limited.

2.3 Key point

The granularity of parallelism must be appropriate for both the underlying hardwares resources and the solutions particular needs

- Fine-grained parallelism needs very fast interaction / communication, e.g. hardware shared memory
- Programs with many dependencies will tend to be more fine-grained
- Course granularity is good when the cost of interaction is high

2.4 Reducing Granularity

Batching is performing work as a group

- E.g. rather than sending each element of an array individually **send all of the required elements together**
- Or rather than a thread getting one small task at a time from a **queue get several tasks at once**

Batching makes computation more **coarse-grained** by **reducing the frequency** of interactions. But only makes sense if there are still enough chunks of work **for all the processors**, and the individual tasks **dont have dependencies** with other tasks.

2.5 Increasing Granularity

Over-dividing the work into more, smaller, units makes computation more **fine-grained**, since interaction is needed for (at least) every unit of work. But can make it easier to **keep all processors busy** (e.g. with its own queue of several small jobs). Especially useful if units of work are **variable or unpredictable in size** since it is hard to divide the work evenly between processors

3 Locality

Fast programs tend to maximize the number of local memory references and minimize the number of non-local memory references.

- Every non-local memory reference **requires communication**, which is an overhead
- Reducing non-local references also **reduces dependences**
- More locality is basically always good, but it **may have a cost**, e.g. more memory or computing

3.1 Increasing locality by using More Memory

Using extra memory can increase parallelism by reduced false dependencies.Privatisation: rather than threads competing to access a single shared variable, **give each thread its own separate copy** that can be used independently

3.2 Increasing locality through Redundant Computation

In a redundant computation each thread **calculates the same value locally**, instead of calculating it once in one thread and communicating the value to each thread. If each thread cannot make progress until it has the value then it **may as well spend the time calculating it for itself**. The communication is no longer required So each thread gets the value sooner And the dependency between threads is removed.

4 Performance strategies

5 Memory references

Reference section

placeholder