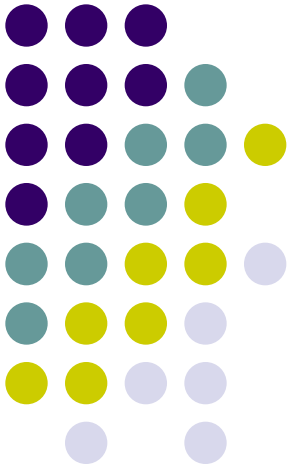
ITMC403 Parallel and Distributed Computing

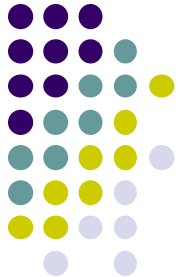
Interconnected Multiprocessor architectures'

-Types

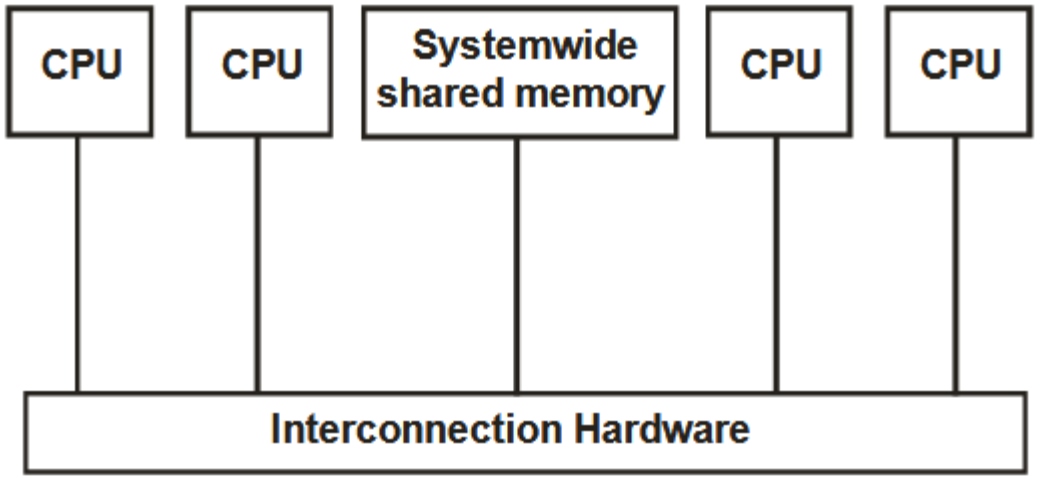
- Vector Parallelism



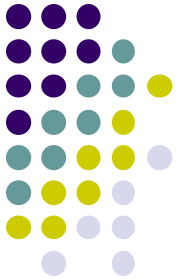
Tightly coupled Interconnected Multiprocessor architectures



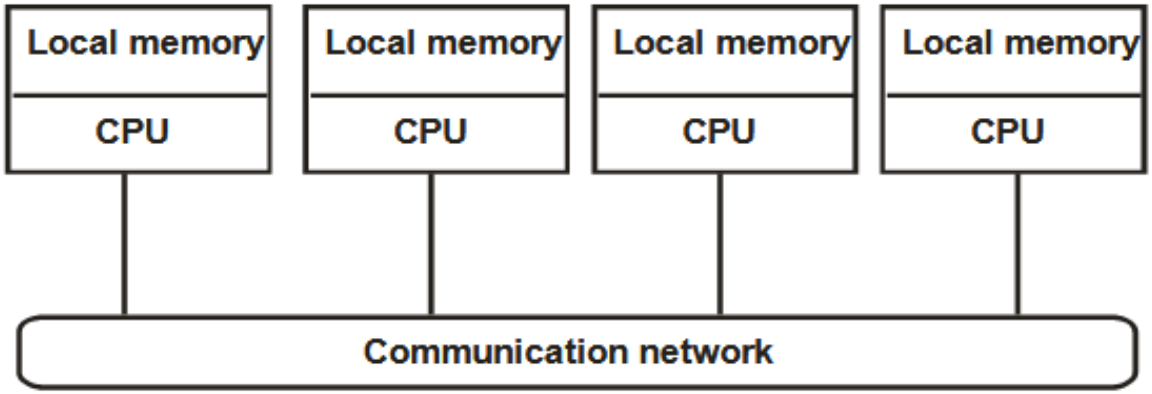
- **In these architectures,** there is a single system wide primary memory (address space) that is shared by multiprocessors, as shown in the figure.
- **For example,** If any processor writes, the value 100 to the memory location x, any other processor subsequently reading from location x will get the value 100. Therefore, in these architectures, any communication between the processors usually takes place through the shared memory.



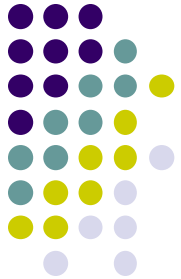
Loosely coupled Interconnected Multiprocessor architectures



- **In these architectures,** the processors do not share memory, and each processor has its own local memory, As shown in the figure.
- **For example:** If a processor writes the value 100 to the memory location x, this write operation will only change the contents of its local memory and will not affect the contents of the memory. In these architectures, all physical communication between the processors is done by passing messages across the network that interconnects the processors.

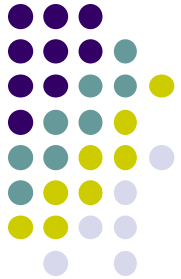


In Class Discussions on both types



- **Tightly coupled architecture** are referred to as **parallel processing systems**, and
- **loosely coupled architecture** are referred to as **distributed computing systems**, or simply distributed systems.
- In contrast to the **Tightly coupled architecture** , the processor of **distributed computing systems** can be located far from each other to cover a wider geographical area.
- In **Tightly coupled architecture**, the number of processors that can be usefully deployed is usually small and limited by the bandwidth of the shared memory. This is not the case with **distributed computing systems** that are more freely expandable and can have an almost unlimited number of processors.

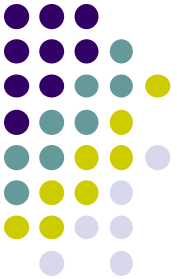
In Class Discussions on both types



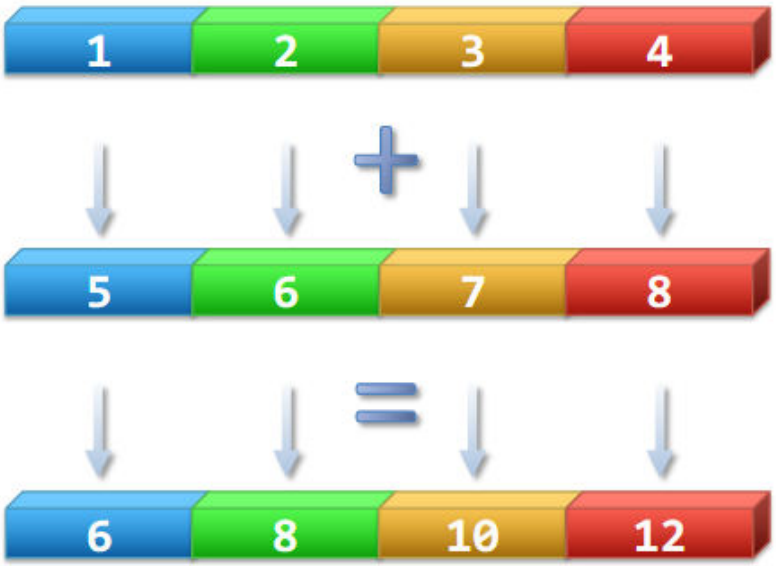
- In short, **loosely coupled architecture**
  - is basically a collection of processors interconnected by a communication network in which each processor has its own local memory and other peripherals, and the communication between any two processors of the system takes place by message passing over the communication network.

  - For a particular processor, its own resources are local, whereas the other processors and their resources are remote. Together, a processor and its resources are usually referred to as a node or site or machine of the distributed computing system.

Vector Parallelism on Multi Processors (SIMD)

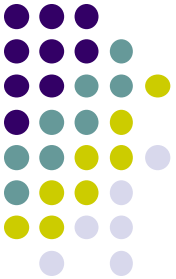


- How It Works, Conceptually



- **Goal:** parallelize computations on vector arrays
  – Line up operands, execute one op on all simultaneously

Three Ways to Look at Vectorization

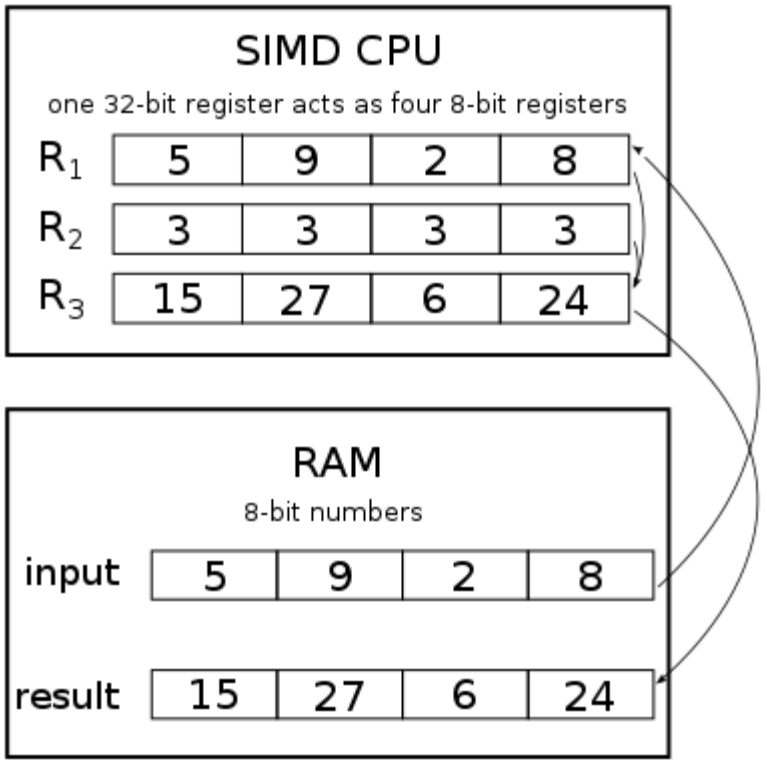


- **Hardware Perspective:** Run vector instructions involving special registers and functional units that allow in-core parallelism for operations on arrays (vectors) of data.
- **Compiler Perspective:** Determine how and when it is possible to express computations in terms of vector instructions.
- **User Perspective:** Determine how to write code with SIMD in mind; e.g., in a way that allows the compiler to deduce that vectorization is possible.
- **How Do You Get Vector Speedup?**
  - Relatively “easy” for user, “challenging” for compiler.
  - Compiler may need some guidance through directives.
  - Programmer can help by using simple loops and arrays.

Hardware Perspective:

- This involves lining up two vectors (R1 and R2), and multiplying their individual elements together to produce vector (VR3).

- **Example:**
- 1 load
- 1 multiply
- 1 save



Compiler Perspective

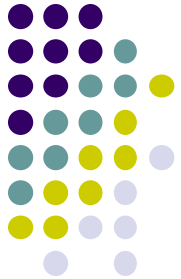
- Think of vectorization in terms of **loop unrolling**
-
  – Unroll by 4 iterations, if 4 elements fit into a vector register

```
for (i=0; i<N; i++) {
        c[i]=a[i]+b[i];
}

for (i=0; i<N; i+=4) {
        c[i+0]=a[i+0]+b[i+0];
        c[i+1]=a[i+1]+b[i+1];
        c[i+2]=a[i+2]+b[i+2];
        c[i+3]=a[i+3]+b[i+3];
}
```

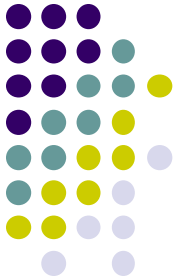
```
Load a(i..i+3)
Load b(i..i+3)
Do 4-wide a+b->c
Store c(i..i+3)
```

Loops That the Compiler Can Vectorize



- **Basic requirements of vectorizable loops:**
  - Number of iterations is known on entry
    – No conditional termination (“break” statements, while-loops)
  - Single control flow; no “if” or “switch” statements
    – Note, the compiler may convert “if” to a masked assignment!
  - Must be the innermost loop, if nested
    – Note, the compiler may reorder loops as an optimization!
  - No function calls but basic math: pow(), sqrt(), sin(), etc.
    – Note, the compiler may inline functions as an optimization!
  - All loop iterations must be independent of each other

User Perspective

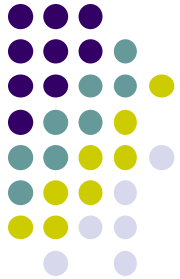


- User's goal is to supply code that runs well on hardware

  - **Thus, you need to know the hardware perspective**
    – Think about how instructions will run on vector hardware
    – Try also to combine additions with multiplications
    – Furthermore, try to reuse everything you bring into cache!

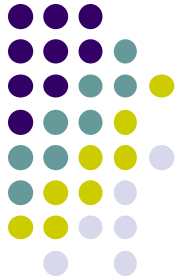
  - **And you need to know the compiler perspective**
    – Look at the code like the compiler looks at it
    – At a minimum, set the right compiler options!

Vector-Aware Coding



- Know what makes codes vectorizable at all
  – The “for” loops (C) or “do” loops (Fortran) that meet constraints
- Know where vectorization ought to occur
- Arrange vector-friendly data access patterns (unit stride)
- Study compiler reports: do loops vectorize as expected?
- Evaluate execution performance: is it near the roofline?
- Implement fixes: directives, compiler flags, code changes
  – Remove constructs that hinder vectorization
  – Encourage/force vectorization when compiler fails to do it
  – Engineer better memory access patterns

Challenge: Loop Dependencies



- Vectorization changes the order of computation compared to sequential case
  - Groups of computations now happen simultaneously

- Compiler must be able to prove that vectorization will produce correct results

- Key criterion: “unrolled” loop iterations must be independent of each other
  - Wider vectors means that more iterations must be independent
  - Note, not all kinds of dependencies are detrimental

- Compiler performs dependency analysis and vectorizes accordingly
  - It will make conservative assumptions about dependencies, unless guided by directives

Loop Dependencies: Read After Write

- Consider adding the following vectors in a loop, N=5:

a = {0,1,2,3,4}
b = {5,6,7,8,9}

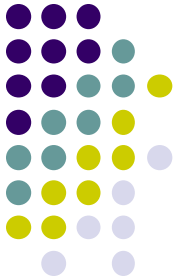
```
for(i=1; i<N; i++)
        a[i] = a[i-1] + b[i];
```

- Applying each operation sequentially:

a[1] = a[0] + b[1] → a[1] = 0 + 6 → a[1] = 6
a[2] = a[1] + b[2] → a[2] = 6 + 7 → a[2] = 13
a[3] = a[2] + b[3] → a[3] = 13 + 8 → a[3] = 21
a[4] = a[3] + b[4] → a[4] = 21 + 9 → a[4] = 30

a = {0, 6, 13, 21, 30}

Loop Dependencies: Read After Write



- Consider adding the following vectors in a loop, N=5:

a = {0,1,2,3,4}
b = {5,6,7,8,9}

```
for(i=1; i<N; i++)
        a[i] = a[i-1] + b[i];
```

- Applying each operation sequentially:

a[1] = a[0] + b[1] → a[1] = 0 + 6 → a[1] = 6
a[2] = a[1] + b[2] → a[2] = 6 + 7 → a[2] = 13
a[3] = a[2] + b[3] → a[3] = 13 + 8 → a[3] = 21
a[4] = a[3] + b[4] → a[4] = 21 + 9 → a[4] = 30

a = {0, 6, 13, 21, 30}

Loop Dependencies: Read After Write

- Consider adding the following vectors in a loop, N=5:

a = {0,1,2,3,4}
b = {5,6,7,8,9}

```
for(i=1; i<N; i++)
    a[i] = a[i-1] + b[i];
```

- Applying vector operations, i={1,2,3,4}:

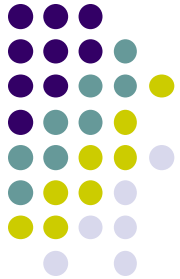
a[i-1] = {0,1,2,3} (load)
b[i] = {6,7,8,9} (load)

{0,1,2,3} + {6,7,8,9} = {6, 8, 10, 12} (operate)

a[i] = {6, 8, 10, 12} (store)

a = {0, 6, 8, 10, 12} ≠ {0, 6, 13, 21, 30} NOT VECTORIZABLE

Loop Dependencies: Synopsis



- **Read After Write**
  - Also called **"flow" dependency** *for(i=0; i<N; i++)*
  - Variable written first, then read *a[i] = a[i-1] + b[i];*
  - Not vectorizable

- **Write After Read**
  - Also called **"anti" dependency** *for(i=0; i<N; i++)*
  - Variable read first, then written *a[i] = a[i+1] + b[i];*
  - Vectorizable

Loop Dependencies: Synopsis

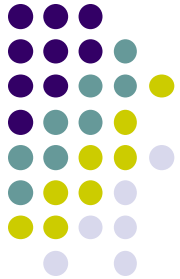
- **Read After Read**
  - Not really a dependency
  - Vectorizable

```
for(i=0; i<N; i++)
    a[i] = b[i%2] + c[i];
```

- **Write After Write**
  - a.k.a “output” dependency
  - Variable written, then re-written
  - Not vectorizable
  - Exception: array sums and products (+=, *=) are vectorizable

```
for(i=0; i<N; i++)
    a[i%2] = b[i] + c[i];
```

Loop Dependencies: Aliasing



- In C, pointers can hide data dependencies!
  - The memory regions that they point to may overlap
- Is this vectorizable?

```
void compute(double *a, double *b, double *c) {
        for (i=1; i<N; i++) {
                a[i] = b[i] + c[i];
        }
}
```

  - ...Not if we give it the arguments *compute(a,a-1,c)*
  - In effect, *b[i]* is really *a[i-1]* → Read After Write dependency
- Compilers can usually cope, at some cost to performance