Implicit and Explicit

Parallel Languages For High Performance Computing

Basic approaches

Basically, there are three approaches to programming high performance computers:

- take an **existing language** and let all the work been done by the compiler;
- **extend an existing** (sequential) language with new constructs to represent parallelism;
- **design a completely new language** that incorporates parallel features.

The first approach

- Is called the **implicit** approach.
- From a user perspective this would be ideal.
- Existing codes could run immediately on high performance platforms, without users having to worry about changing and optimising programs.
- This can lead to substantial savings in development costs and is also very attractive for vendors of high performance computers.
- Although there has definitely been some success in this area, there are some problems.

The first approach (Problems)

The second problem: The full parallel potential of the problem cannot always be exploited. This is due to the fact that certain information known to the programmer is lost when coding the problem in a computer language or that the programmer has introduced <u>fake dependencies</u> in the program, which inhibit the compiler from doing the necessary transformations.

The second problem: is related to the architectural characteristics of the underlying system. The performance of a chosen algorithm <u>strongly depends on the architecture</u> on which the algorithm is executed. Many of the current algorithms are based on the characteristics of computer systems designed a decade or more ago. These systems had small memories and hence algorithms were favoured which put a limited burden on the memory capacity. Storage capacity of current 1 memories is not effectively used by these old algorithms. So, to profit from the advances in hardware technology, it turns out that these algorithms need to be replaced by newer algorithms.

The second approach

- Is called the **explicit** approach.
- existing languages are extended with new language constructs as to express parallelism.
- The clear advantage of this approach is that users have already been trained in using the base languages.
- To get their programs running efficiently, only a limited set of extra functions need to be applied.

- **The problems with this approach:**
  - Concentrate on the sometimes difficult interaction between the base sequential language and the parallel features (debugging).
  - Moreover, there is a lack of standardisation, meaning that a number of extensions has been developed with similar functionality, but different appearance.

The third approach

- The last approach is to develop a new language as to present a coherent approach to programming for high performance machines.

- In designing a new language one can choose an implicit, an explicit, or a hybrid approach.

- **The problems with this approach:**
  - This approach clearly implies rewriting of the application software, which can be very costly and has many risks as well.

The first approach (example)

- **Compiler-based detections and transformations**
- The basic instructions are defined in terms of operations on vectors instead of scalars.
- languages like FORTRAN 77 the basic operations are defined on scalars.
- To use these vector computers efficiently, the scalar operations in a program written in such languages must be transformed into vector operations.
- To give a simple example,

the program	transformation

DO i = 1, 3
    a(i) = b(i) + c(i)
END DO
	a(1:3)=b(1:3)+c(1:3)

- In vector computers, this operation is implemented by loading both b(1:3) and c(1:3) from memory, feeding them one element after the other to a pipelined adder and getting – after a few letancy– the results out element by element, which are then finally stored back in memory. In vector computers, the data elements are processed in a stream by the operational units.

The second approach (example)

- An alternative way of getting speed up is to use a parallel computer. In a parallel computer, each processor is executing a part of the program. To be able to use a parallel computer, a compiler must identify which parts of a program can be executed in parallel. It is easy to see that each body instance of the above loop can be independently executed. Hence, each index can be assigned to a different processor without any order restrictions.
- To indicate the (possible) parallel execution of a loop, the keyword DO can be replaced by the keyword PAR DO, indicating that the iterations can be processed independently

the program hgfhjlkm r	Updated program

DO i = 1, 3
    a(i) = b(i) + c(i)
END DO
	PAR DO i = 1, 3
	    a(i) = b(i) + c(i)
	END DO

- We could assign a(1)=b(1)+c(1) to Processor 1, a(2)=b(2)+c(2) to Processor 2, etc., but any other order is also allowed.

Computation strategies

- **Lazy evaluation (call-by-need)**
- is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing).
- Example:

  fun g(x,y,z) = if x<2 then y+3 else z+6

  Depending on the value of *x*, either the value of *y* or the value of *z* will be required, but not both.

  Computation is demand-driven: we only compute as much of the intermediate computation as we need to get the result.

  Require analysing of the program

Computation strategies

- **Eager evaluation (call-by-value)**

- Eager evaluation means expression is evaluated as soon as it is encountered.
- Also, named as a strict evaluation or a greedy evaluation,

- All arguments to a function or operator are evaluated before the function is applied

  e.g.: (square (a+ (b* 2)))

Models of parallel computation: a systems view

- **SD** : Single Data space (no distribution)
- **MD** : Multiple Data spaces (distribution)
- **SP** : Single Program
- **MP** : Multiple (different) Programs
- ***** : Replication operator (multiple copies)

A specific model of a parallel system

- Any specific model of a parallel system is then defined by combining a program and a data space qualifier and an optional *.
  - The data space qualifier indicates if data is considered to reside in a single common memory (such as in shared-memory systems) or is distributed among logically different memories (such as in distributed-memory systems).
  - The program qualifier tells us if the model allows for a single program or (functionally) different programs in execution.
  - The * qualifier indicates if multiple copies of programs or data are allowed

A specific model of a parallel system (examples)

- Examples:
  - a sequential model is defined as **SPSD** in this system.
  - A model defined as **MPSD**, allows the creation of multiple, functionally different programs acting on a single, non-distributed data space.
  - Another classification we will encounter is **SP*MD**, meaning multiple, identical programs acting on multiple data spaces.
  - Along these lines SPMD means a single program acting on multiple data spaces.
  - The classical **SIMD** computer, such as the CM-2, conforms to this model, where a single program residing in a single controller acts on multiple (simple) processing elements, each storing a single data element