5. Compare between tightly coupled Multiprocessor and Loosely coupled systems.

Sr. No.	Loosely Coupled Multiprocessor System.	Tightly coupled Multiprocessor system.
1	There is distributed memory in loosely coupled multicomputers system.	There is shared memory in tightly coupled multicomputer system.
2	Loosely coupled multicomputers system has low data rate.	Tightly coupled multicomputers system has high data rate.
3	Loosely coupled multicomputers systems has low degree of interaction between tasks.	Loosely coupled multicomputers systems has high degree of interaction between tasks.
4	In loosely coupled multicomputers systems there is direct communication between processor and I/O devices.	In tightly coupled multicomputers system there is IOPIN to help communication between the processor and I/O devices.
5	The cost of loosely coupled multicomputers systems is less.	The cost of loosely coupled multicomputer system is more costly.
6	In loosely coupled multicomputers system, Memory conflict don't take place.	In tightly coupled multicomputers system have memory conflict.
7	Application of loosely coupled multicomputers system are in distributed computing system.	Application of tightly coupled multicomputers system are in parallel processing systems.
8.	In loosely coupled multicomputers system, modules are connected through message transfer system network.	In tightly coupled multicomputers system there is PMIN, IOPIN and ISIN network.

6.2) PARALLEL LANGUAGES AND COMPILERS

6.2.1) Language Features for Parallelism: -

Chang and Smith (1990) classified the language features for parallel programming into six categories according to functionality. These features are idealized for general-purpose applications. In practice, the real languages developed or accepted by the user community might have some or no features in some of the categories. Some of the features are identified with existing language/compiler development. The listed features set guidelines for developing a user-friendly programming environment.

1) **Optimization Feature:** These features are used for program restructuring and compilation directives in convening sequentially coded programs into parallel forms. These purpose is to match the software parallelism with the hardware parallelism in the target machine.
   - Automated parallelizer -Examples are: Express C automated parallelizer and the Alliant FX Fortrancompiler.
   - Semiautomated parallelizer—Needs compiler directives or programmer's interaction, such as DINO.
   - Interactive restructure support—Static analyzer, run-time statistics, dataflow graph, and code translator for restructuring FORTRAN code, such as the MIMDizer from Pacific Sierra.

2) **Availability Feature:** These are features that enhance the user-friendliness, make the language portable to a large class of parallel computers, and expand the applicability of software libraries.
   - Scalability- The language is scalable to the number of processors available and independent of hardware topology.
   - Compatibility- The language is compatible with an established sequential language.
   - Portability - The language is portable to shared-memory multiprocessors, message-passing multicomputer, or both.

3) **Synchronization/Communication Feature:** Listed below are desirable language features for synchronization or for communication purposes:
   - Single-assignment languages.
   - Shared variables (locks) for IPC.
   - Logically shared memory such as the tuple space in Linda.
   - Send/receive for message passing.
   - Rendezvous in Ada.
   - Remote procedure call.

- Dataflow languages such as Id.
- Barriers. Mailbox, semaphores, monitors.

4) **Control of Parallelism**: Listed below are features involving control constructs for specifying parallelism in various forms:
   - Coarse, medium, or fine grain.
   - Explicit versus implicit parallelism.
   - Global parallelism in the entire program.
   - Loop parallelism in iterations.
   - Task-split parallelism.
   - Shared task queue.
   - Divide-and-conquer paradigm.
   - Shared abstract data types.
   - Task dependency specification.

5) **Dara Parallelism Feature:**
   Data parallelism is used to specify how data are accessed and distributed in either SIMD or MIMI) computers.
   - Run-time automatic decomposition- Data are automatically distributed vvitl1 no user intervention, as in Express.
   - Mapping specification- Provides facility for users to specify communication patterns or how data and processes are mapped onto the hardware, as in DINO.
   - Virtual processor support - The compiler maps the virtual processors dynamically or statically onto the physical processors, as in PISCES 2 and DINO.
   - Direct access to shared data- Shared data can be directly accessed without monitor control, as in Linda.
   - SPMD (single program multiple data) support—SPMD programing, as in DINO and Hypertasking.
6) **Process Management Feature:** These features are needed to support the efficient creation of parallel processes, implementation of multithreading or multitasking, program partitioning and replication, and dynamic load balancing at run lime.
   - Dynamic process creation at run time.
   - Lightweight processes (threads) - Compared to UNIX (heavyweight) processes.
   - Replicated workers- Same program on every node with different data (SPMD mode)
   - Partitioned networks- Each processor node might have more than one process and all processor nodes might run different processes.
   - Automatic load balancing—The workload is dynamically migrated among busy and idle nodes to achieve the same amount of work at various processor nodes

6.3) Code Optimization and Scheduling: -

In this section, we describe the roles of compilers in code optimization and code generation for parallel computers. In no case can one expect production of a true optimal code which matches the hardware behavior perfectly. Compilation is a software technique which transforms the source program to generate better object code, which can reduce the running time and memory requirement. On a parallel computer, program optimization often demands an effort from both the programmer and the compiler.

6.3.1) Scalar Optimization with Basic Blocks: -

a. Instruction scheduling is often supported by both compiler techniques and dynamic scheduling hardware. In order to exploit instruction-level parallelism (ILP), we need to optimize the code generation and scheduling process under both machine and program constraints. Machine constraints are caused by mutually exclusive use of functional units, registers, data paths, and memory. Program constraints are caused by data and control dependences. Some processors, like those with VLIW architecture, explicitly specify ILP in their instructions. Others may use hardware interlock, out-of-order execution, or speculative execution. Even machines with dynamic scheduling hardware can benefit from compiler scheduling techniques.

b. There are two alternative approaches to supporting instruction scheduling. One is to provide an additional set of nontrapping instructions so that the compiler can perform aggressive static instruction scheduling. This approach requires an extension of the instruction set of existing processors. The second approach is to support out-of-order execution in the micro-architecture so that the hardware can perform aggressive dynamic instruction scheduling. This approach usually does not require the instruction set to be modified but requires complex hardware support.

c. In general, instruction scheduling methods ensure that control dependences, data dependences, and resource limitations are properly handled during concurrent execution. The goal is to produce a schedule that minimizes the execution time or the memory demand, in addition to enforcing correctness of execution.

d. Static scheduling at compile time requires intelligent compilation support, whereas dynamic scheduling at rim time requires sophisticated hardware support. In practice, dynamic scheduling can be assisted by static scheduling in improving performance.

Precedence Constraint: -

a. Speculative execution requires the use of program profiling to estimate effectiveness. Speculative exceptions must not terminate execution. In other words, precise exception handling is desired to alleviate the control dependence problem. The data dependence problem involves instruction ordering and register allocation issues.

b. If a flow dependence is detected, the write must proceed ahead of the read operation involved. Similarly, output dependence produces different results if two writes to the same location are executed in a different order. Antidependence enforces a read to be ahead of the write operation involved. We need to analyze the memory variables. Scalar data dependence is much easier to detect. Dependence among arrays of data elements is much more involved. Other difficulties lie in Interprocedural analysis, pointer analysis, and register allocations interacting with code scheduling.

Basic Block Scheduling: - A basic black (or just a block) s a sequence of statements satisfying two properties:

(1) No statement but the first can be reached from outside the block; i.e. there are no branches into the middle of the block.

(2) All statements are executed consecutively if the first one is. Therefore, no branches out or halts are allowed until the end of the block. All blocks are required to be maximal in the sense that they cannot be extended up or down without violating these properties.

For local optimization only, an extended basic block is defined as a sequence of statements in which the first statement is the only entry point. Thus an extended block may have branches out in the middle of the code but no branches into it. The basic steps for constructing basic blocks are summarized below:

(l) Find the leaders, which are the first statements in a block. Leaders are identified as being one or more of the following:

(a) The first statement of the code.

(b) The target of a conditional or unconditional branch.

(c) A statement following a conditional branch.

(2) For a leader, a basic block consists of the leader and all statements following up to but excluding the next leader. Note that the beginning of inaccessible code (dead code) is not considered a leader. In fact, dead code should be eliminated.

6.3.1) Local and Global Optimizations: -

We first describe local code optimization within basic blocks. Then we study global optimizations among basic blocks. Boflt iniraprooodural and interprocedutal optimizations are discussed. Finally, we identify some machine-dependent optimizations. Readers will realize the limitations and potentials of these code optimization methods.

1) Local Optimization: -

These are code optimizations performed only within basic blocks. The information needed for optimization is gathered entirely from a single basic block, not from an extended basic block. No control-flow information between blocks is considered. Listed below are some local optimizations often performed:

(1) Local Common Subexpression Elimination: - If a Subexpression is to be evaluated more than once within a single block, it can be replaced by a single evaluation. For Example |o.s, in block B7, t9 and tl5 each compute 4 "' (j - i}, and 112 and tlii each compute 4 "‘ j. Replacing 115 by t9, and 118 by 112, we obtain the following revised code for B1', which is shorter to execute.

```
tfi := j l
t'-1 := 4 " til
temp := A[t'5l']
tl2 := 4 * j
ti3 := .4.[ti2]
ans] -.=t13
A[tl'.'-'3] := temp
```

(2)Local Constant Folding or Propogorion Sometimes some constants used in instructions can be computed at compile time. This often takes place in the initialization blocks. The compile-time generated constants are then folded to eliminate unnecessary calculations at run time. in other cases, a local copy may be propagated to eliminate unnecessary calculations.

{3} Algebraic Optimization to Si'npl'ifiv Earpressiorrs For example, one can replace the t'derrtr'.l}r statement A :=
B + 0 or A := B * 1 by A := B and later even replace references to this A by references to B. Or one can use the commutative law to combine expressions C := A + B and D := B + A. The associative and di'srri£mrr've
i'aw.'r can also be applied on equal-priority operators, such as replacing (rt — b) + c by ti (b — c) if (b — c) has already been evaluated earlier.
{4} instruction Reordering Code reordering is often practiced to maximize tlte pipeline utilization or to enable overlapped memory accesses. Some orders yield better code than others. Reordered instructions lead to better scheduling, preventing pipeline or memory delays. In the following example, instruction I3 may be delayed in memory accesses:
ll: Load R1, A
12: Load R2, B
13: Add R2, R1, R2 — delayed
14: Load R3, C
With reordering, the instruction 13 may experience no delay:
Load R1, A
Load R2, B
Load R3, CAdd
R2, R1, R2 — not delayed
{5} Elimination of Dead Code or Unorv Operators Code segments or even basic blocks which are not accessible or will never be referenced can be eliminated to save compile time, run time, and spaoe requirements.
GEE:
Unary operators, sueh as arithmetic negation and logical cornplement, can often be eliminated by applying algebraic laws, such as x I {—_v) = x — __\-', —{.t — y] = _t-' — x, (—x] " (—_t-'} = x ' _1-', Not(Not A) = A, etc. Boolean expression evaluation can often be optimized after some form of minimization.
Gabe] Optimizations These are code optimizations performed across basic block boundaries. Controlflow information among basic blocks is needed. John I-lennessy (1992) has classified intraprocedural global optimizations into three types:
{I} Global Versions of local Optimizations These include global common subeitpression elimination, global constant propagation, dead code elimination, etc. The following example fiirtber optimizes the code in
Example 10.6 if some global optimizations are performed.