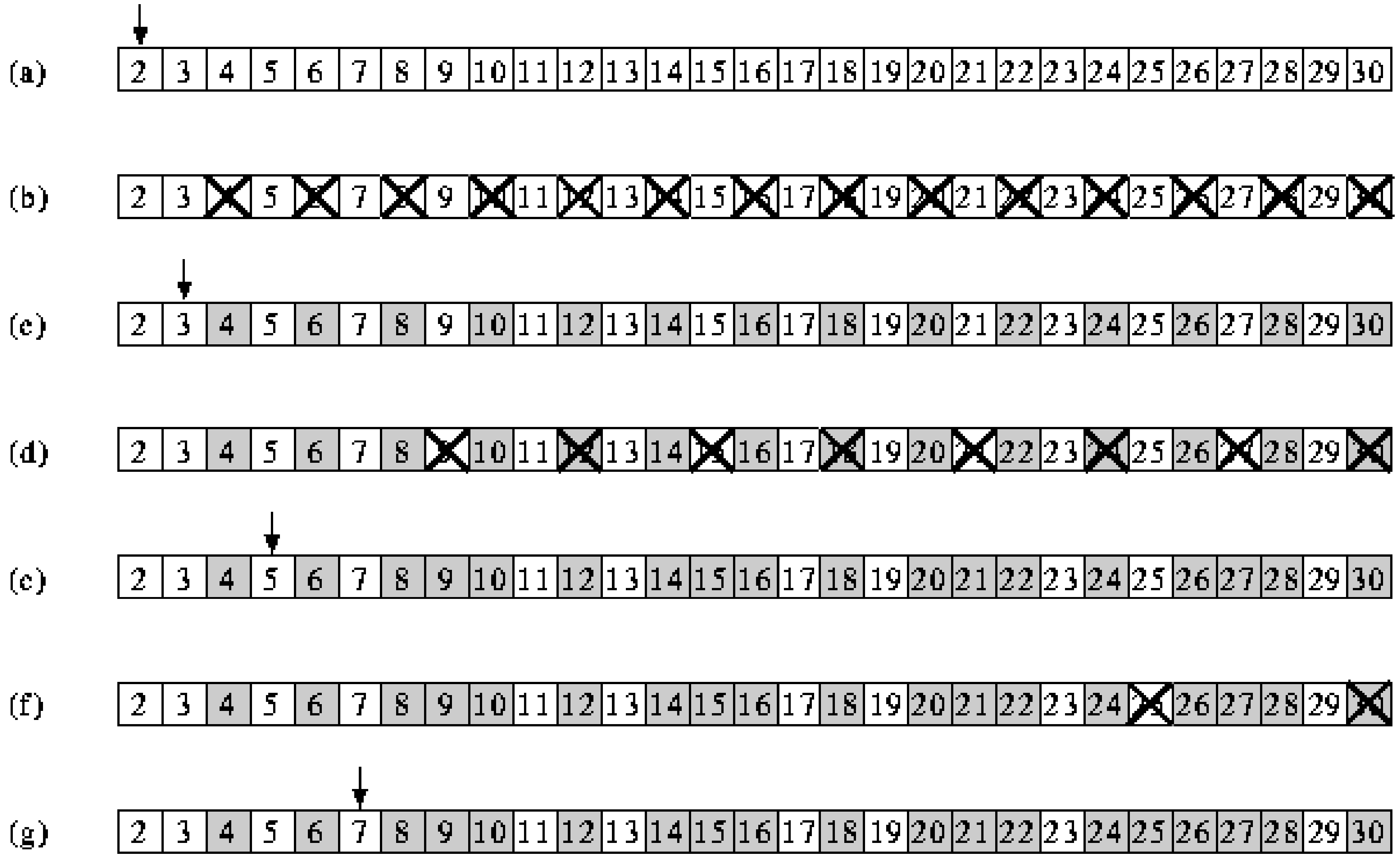
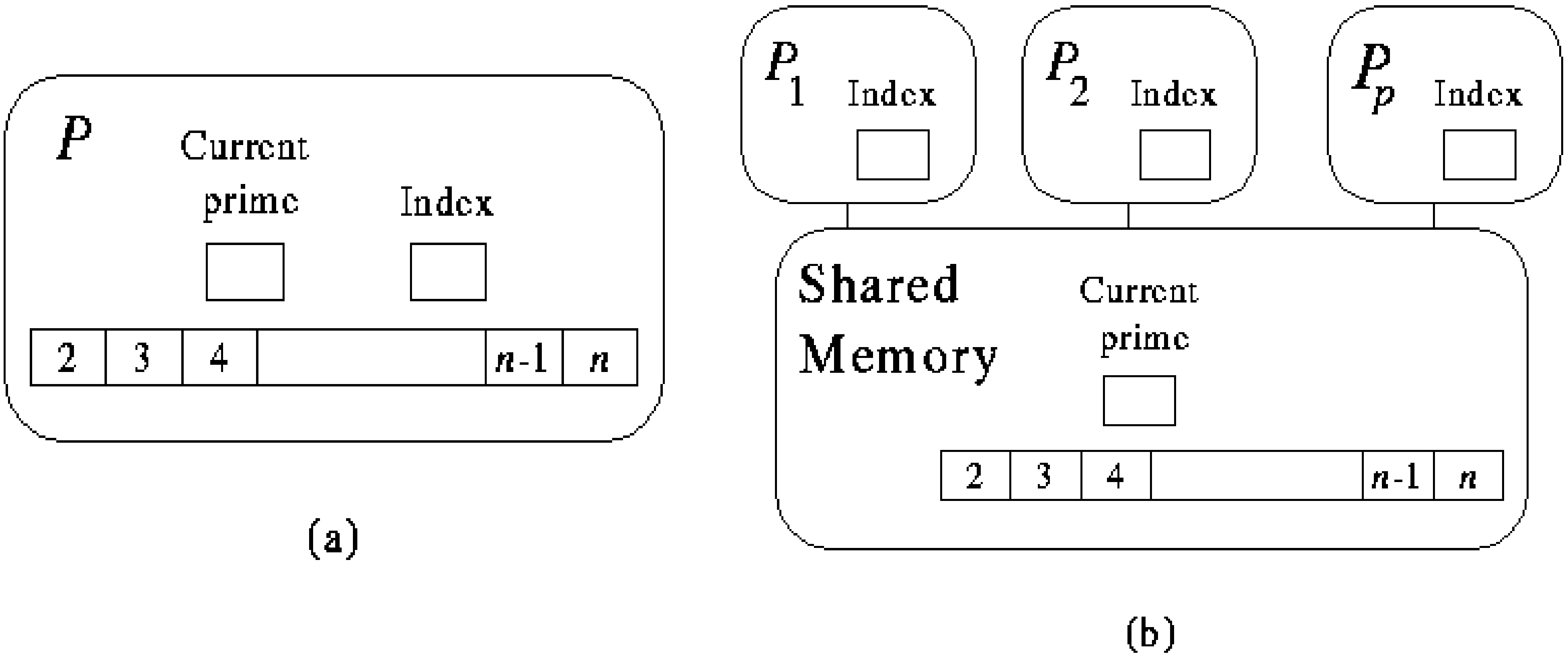
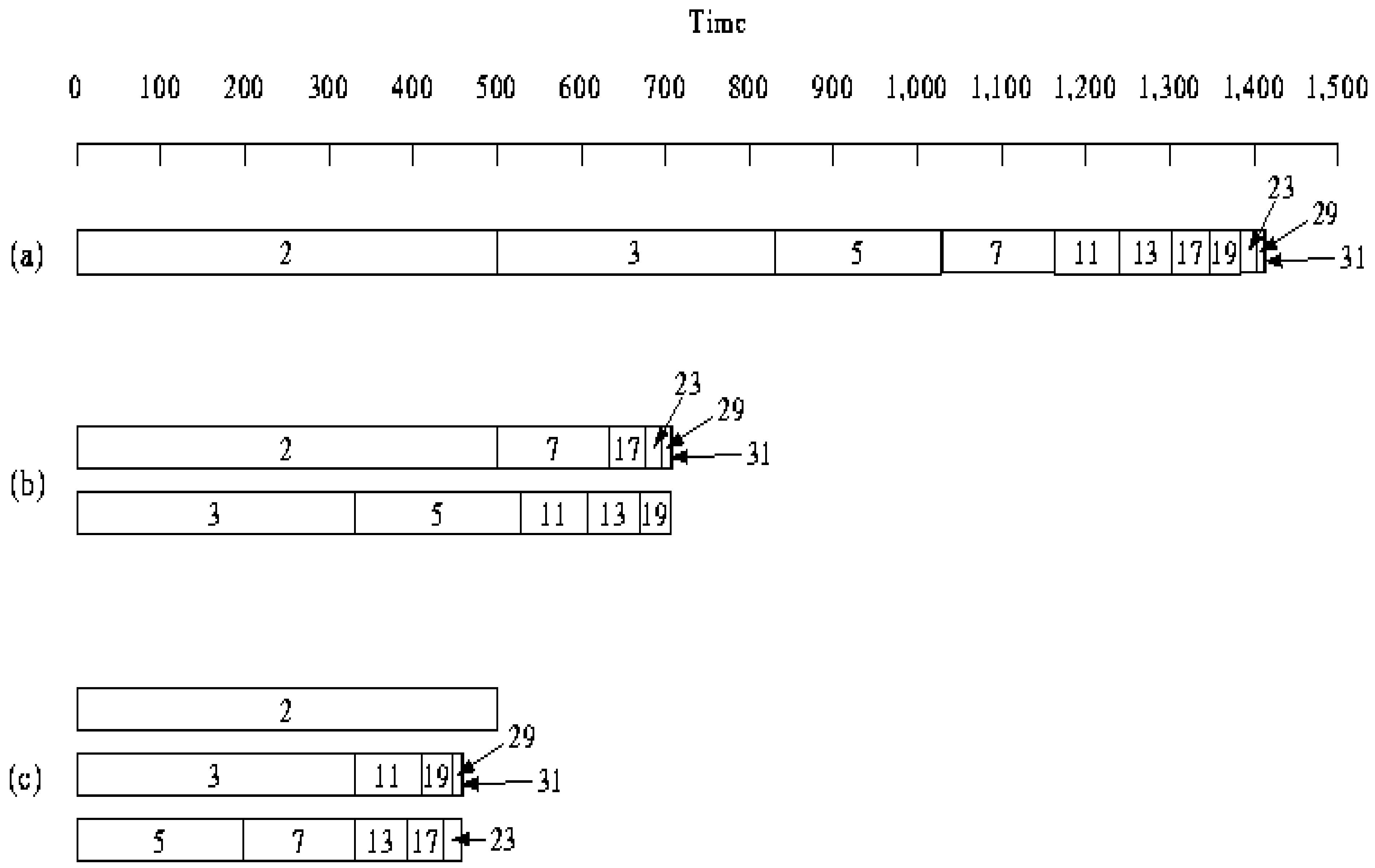
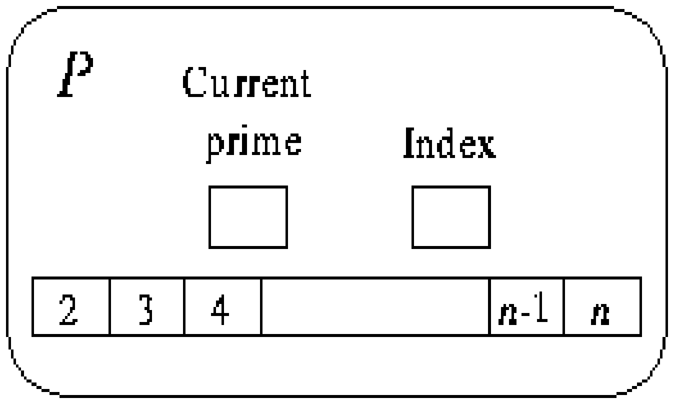
Example of a Parallel Algorithm

Sieve of Eratosthenes

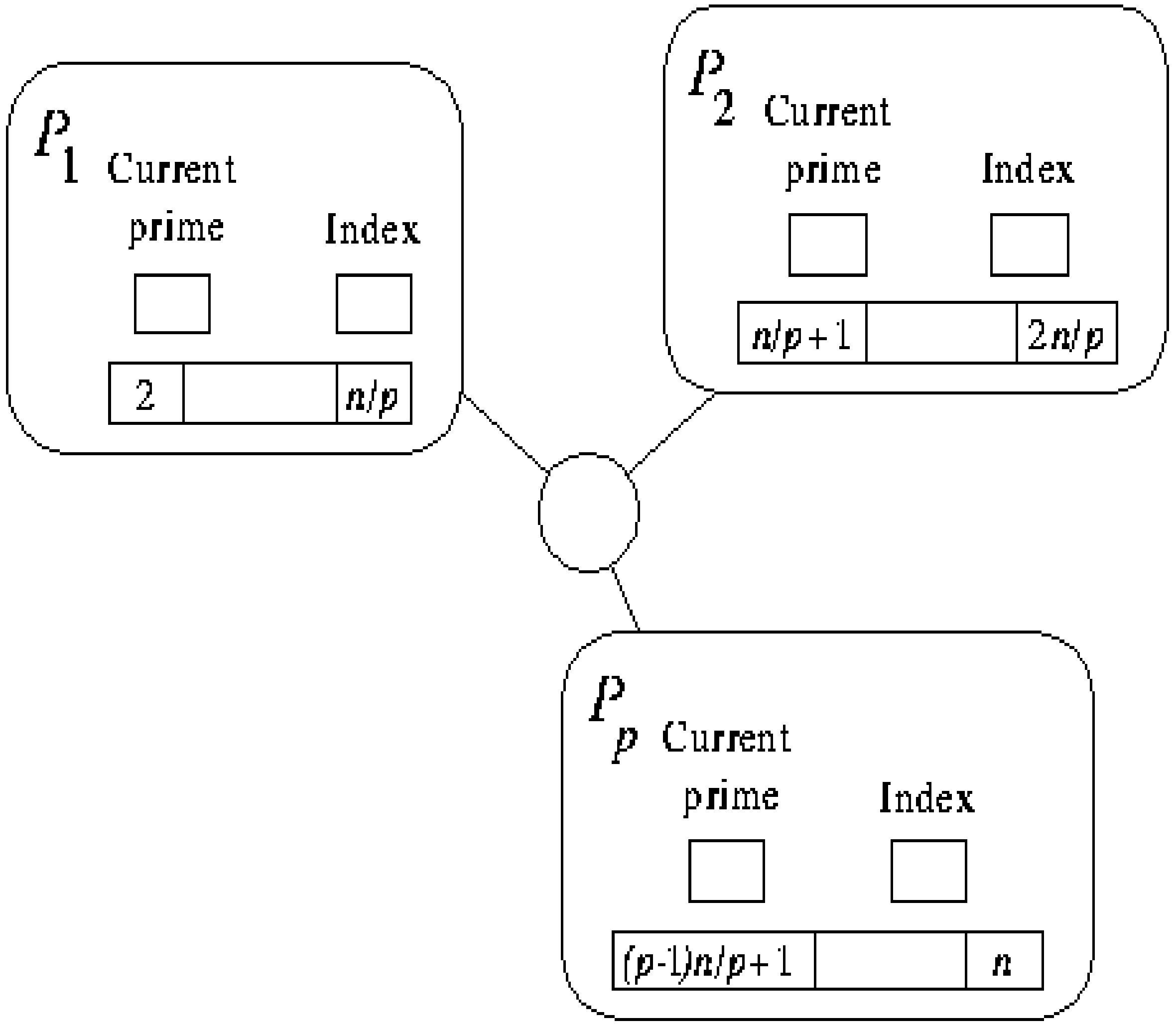



P
Current prime
Index
2
3
4
n-1
n
(a)
P_1
Index
P_2
Index
P_p
Index
Shared Memory
Current prime
2
3
4
n-1
n
(b)

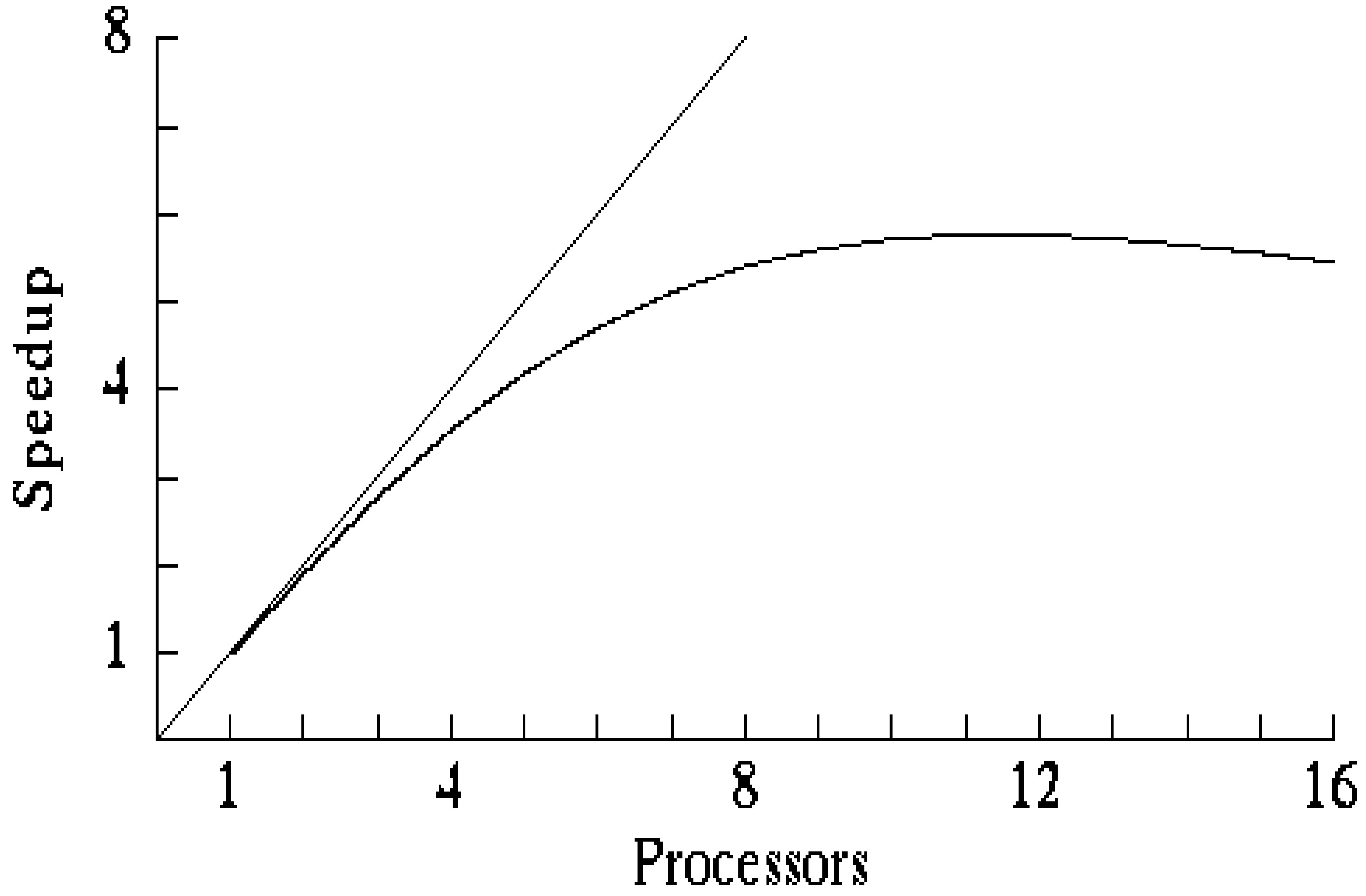

Time
0
100
200
300
400
500
600
700
800
900
1,000
1,100
1,200
1,300
1,400
1,500
(a)
2
3
5
7
11
13
17
19
23
29
31
(b)
2
7
17
23
29
31
3
5
11
13
19
(c)
2
3
11
19
29
31
5
7
13
17
23

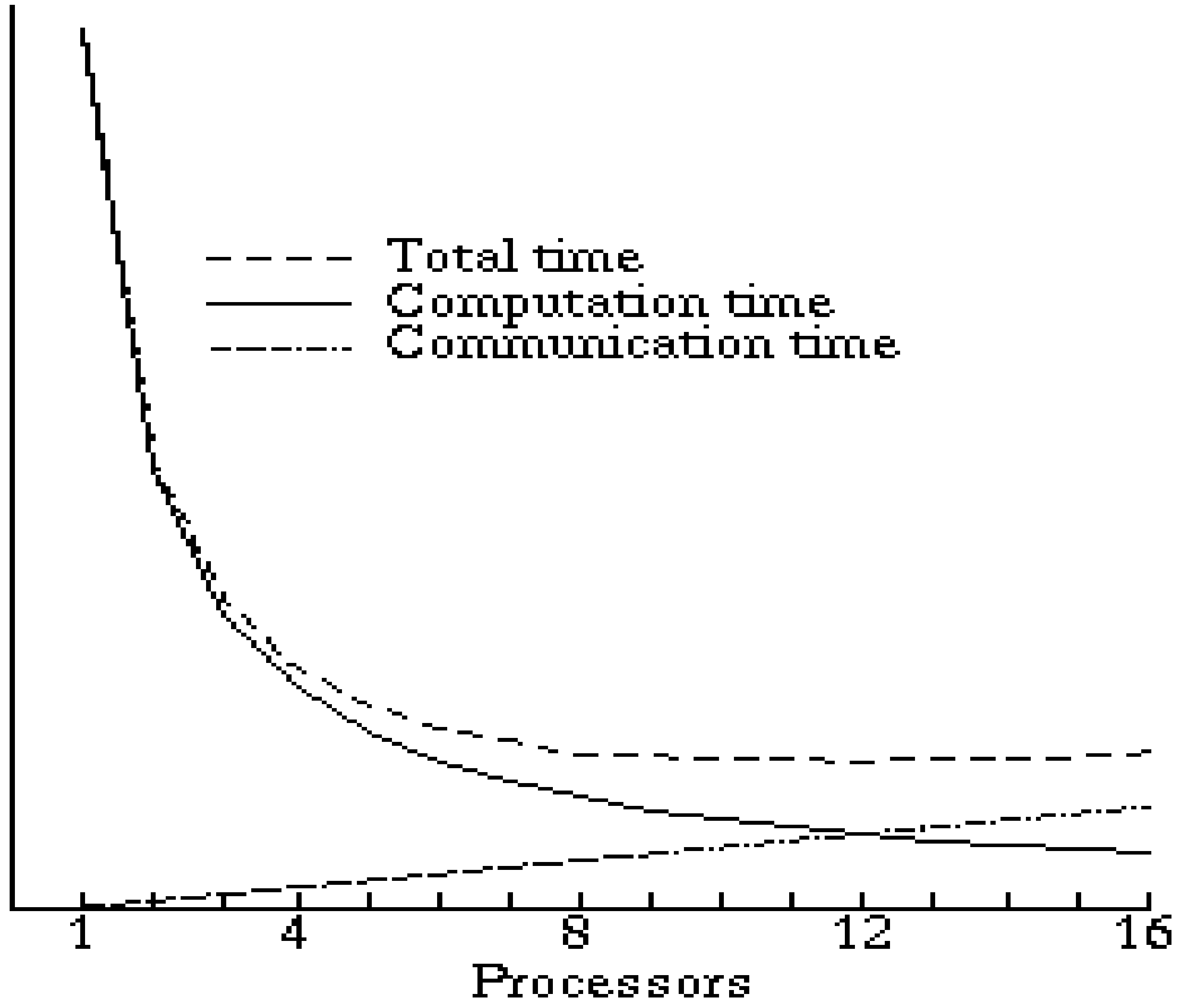

P
Current prime
Index
2
3
4
n-1
n

(a)


P_1
Current prime
Index
2
n/p
P_2
Current prime
Index
$n/p+1$
$2n/p$
P_p
Current prime
Index
$(p-1)n/p+1$
n

(b)


8
4
1
Speedup
1
4
8
12
16
Processors


Total time
Computation time
Communication time
1
4
8
12
16
Processors

SPMD

Advantages

- **Different instructions can be implemented at the same cycle and it has a coarse grain.**
- **Looser synchronization requirement than SIMD.**

Disadvantages

- **Every processor is required to execute the same program, which is not flexible and the degree of parallelism is reduced.**

MPMD

Advantages

- Different processors can execute different programs, thus increasing the degree of parallelism.
- More general than SPMD.
- Can support coarse grain.

Disadvantages

- It is more difficult to balance the load on each processor to minimize idle processor time.

Parallel Programming Overview

- **Parallel software development has lagged far behind the advances of parallel hardware.**
- **The lack of adequate parallel software is the main hurdle to the acceptance of parallel computing by the mainstream user community.**
- **Compared to their sequential counterparts, today's parallel system software and application software are few in quantity and primitive in functionality.**

Why Is Parallel Programming Difficult?

- Parallel programming is a more complex intellectual process than sequential programming.
- It involves all issues in sequential programming, plus many more issues that are intellectually more challenging.
- There are many different parallel programming models.
- Software environment tools such as compiler, debugger, and profiler are much more advanced for sequential programs development.
- More people have been practicing sequential programming than parallel programming.

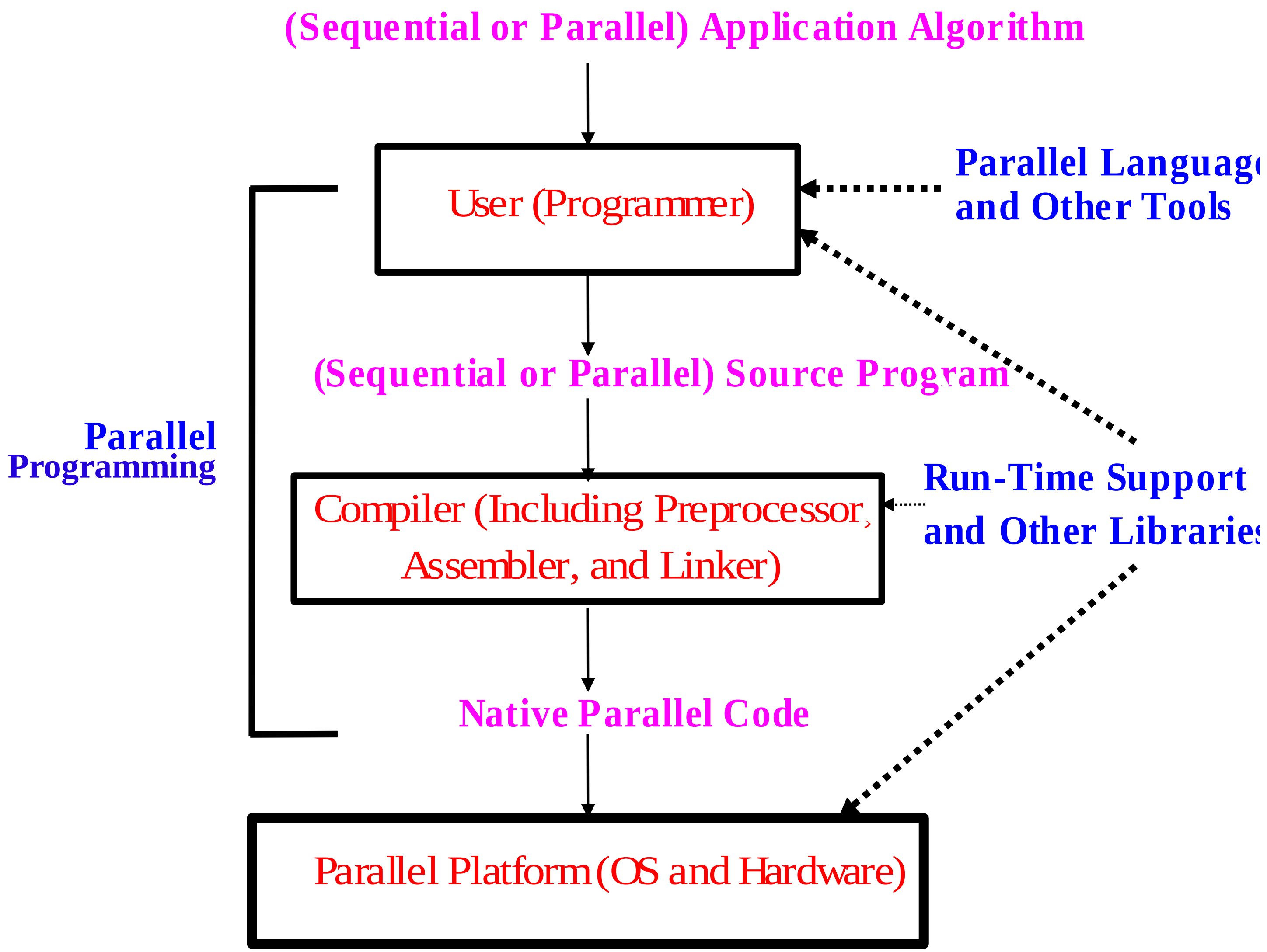
Advances in Parallel Programming

Despite the above pessimistic review, there has been much progress in the parallel programming field. Many parallel algorithms have been developed.

- **The native models are converging toward two models:**
  - **the single-address space, shared-variable model for PVPs, SMPs, and DSMs,**
  - **and the multiple-address space, message-passing model for MPPs and clusters.**
- **The SIMD model is useful for special purpose, embedded applications such as signal, image, and multimedia processing.**
- **A high-performance parallel computer should be viewed as a huge entity with *single system image*.**

Parallel Programming Environments

From a user's viewpoint, a typical parallel processing system has a structure.


(Sequential or Parallel) Application Algorithm
User (Programmer)
Parallel Languag
and Other Tools
(Sequential or Parallel) Source Program
Parallel Programming
Compiler (Including Preprocessor,
Assembler, and Linker)
Run-Time Support
and Other Librarie
Native Parallel Code
Parallel Platform (OS and Hardware)

Environment Tools

Environment tools are the set of tools normally not associated with an operating system or a programming language. Environment tools include the following types:

- ***Job management tools*** are used to schedule system resources and manage user jobs.
- ***Debugging tools*** are used to detect and locate semantic errors in parallel and sequential applications.
- ***Performance tools*** are used to monitor user applications to identify performance bottlenecks, which is also known as *performance debugging*.

Parallel Programming Approaches

There are three means of extension: *library subroutines*, *new language constructs*, and *compiler directives*.

- ***Library Subroutines***: They provide functions to support parallelism and interaction operations. Examples of such libraries include the MPI message passing library and the MPICH, OpenMP, POSIX, Pthreads multithreading library.

- ***New Constructs***: The programming language is extended with some new constructs to support parallelism and interaction. An example is the aggregated array operations in Fortran 90.

- ***Compiler Directives***: The programming language stays the same, but formatted comments, called ***compiler directives*** (or ***pragmas***), are added.

Example

All three parallel programs perform the same computation as the sequential C code.

```
for ( i = 0 ; i < N ; i ++ )  A[i] = b[i] * b[i+1] ;
for ( i = 0 ; i < N ; i ++ )  c[i] = A[i]  + A[i+1] ;
```

(a) A sequential code fragment

```
id = my_process_id () ;
p = number_of_processes () ;
for ( i = id ; i < N ; i = i+p )  A[i] = b[i] * b[i+1] ;
barrier () ;
for ( i = id ; i < N ; i = i+p )  c[i] = A[i]  + A[i+1] ;
```

(b) Equivalent parallel code using library routines

```
A(0:N-1) = b(0:N-1) * b(1:N)
c(0:N-1) = A(0:N-1) + A(1:N)
```

(c) Equivalent code in Fortran 90 using array operations

```
#pragma parallel
#pragma shared ( A, b, c )
#pragma local ( i )
{
#pragama pfor iterate (i=0; N ; 1)
for ( i = 0 ; i < N ; i ++ )  A[i] = b[i] * b[i+1] ;
#pragma synchronize
#pragma pfor iterate (i=0; N ; 1)
for ( i = 0 ; i < N ; i ++ )  c[i] = A[i] + A[i+1] ;
}
```

(d) Equivalent code using pragmas

Comparison

Three Approaches to Implementing Parallel Programming System

Approach	Example	Advantages	Disadvantages
Message-Passing Library	Express, PVM, MPI	Easy to implement, need not a new compiler	Overhead, partitioning required
Language constructs	Fortran90, Cray Craft, Cuda	Allow compiler check, analysis and optimization	Hard to implement, complex compiler
Compiler Directives	HPF, OpenMP, pThread	between a library and language constructs	

The approaches and the programming models can all be combined in various ways on any parallel platform.

Processes, Tasks, and Threads

On a parallel computer, a user application is executed as *processes*, *tasks*, or *threads*.

Definitions of an Abstract Process

A *process* *P* is a 4-tuple *P* = (*P*, *C*, *D*, *S*), where *P* is the *program* (or the *code*), *C* the *control state*, *D* the *data state*, and *S* the *status* of the process *P*.

Program (Code)

Any process is associated with a program. As a concrete example, consider the following C code:

```
main() {
  int i = 0;
  fork(); fork();
  printf("Hello!\n");
}
```

Control and Data States

A program uses two sets of variables: *Data variables* and *Control variables*. The union of these two sets forms the set of *program variables*.

Control variables are program counters.

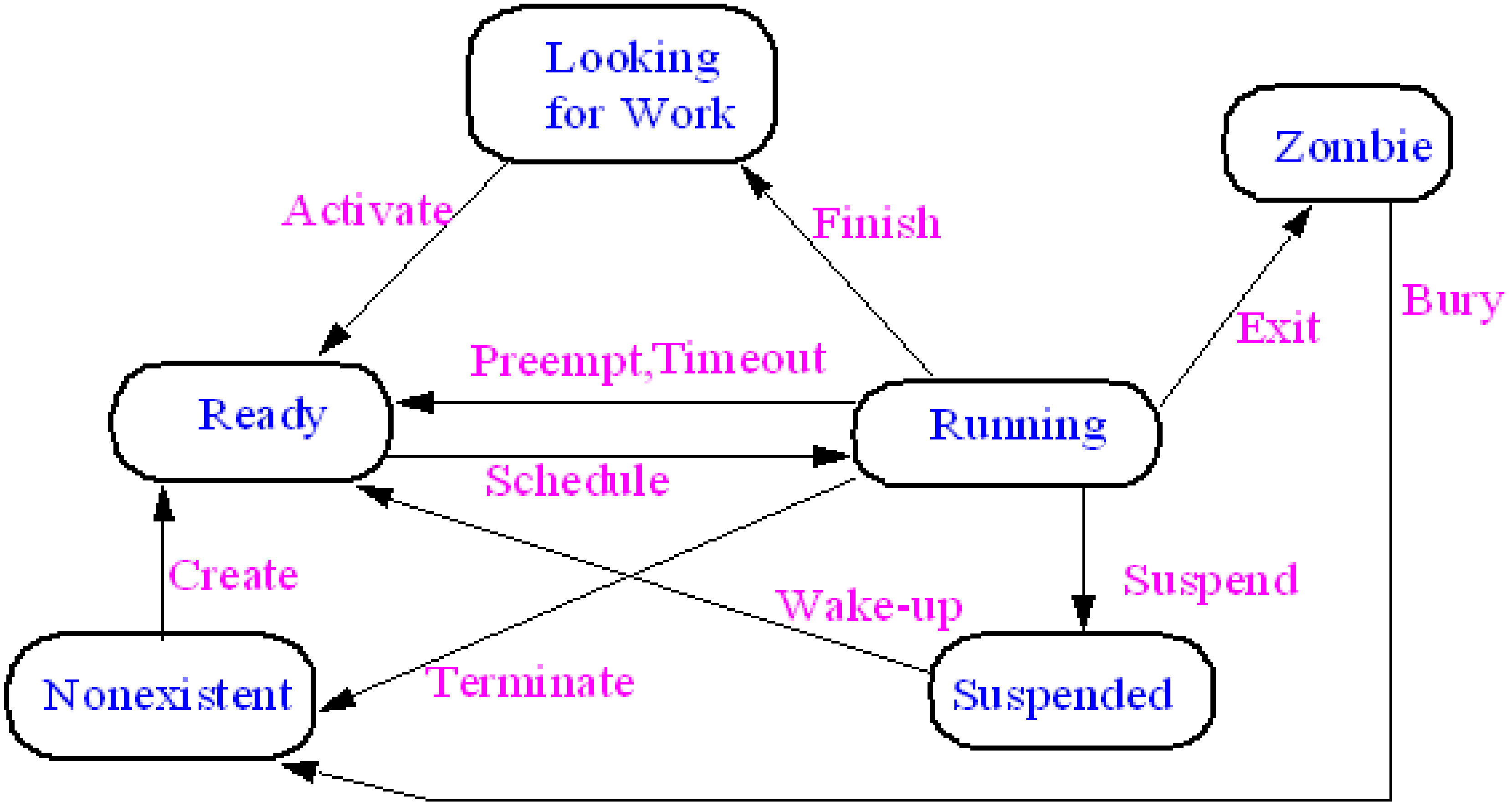
For a process with a single *thread of control*, there is just one control variable: the program counter.

In multiple threads of control, each control variables may hold the program counter value of that thread.

Process Status

A process has a certain *status* at any time.

The process state transition diagram



Another frequently used operation is *process switching*, which refers to transferring a running process to either a suspended or ready status and scheduling the next ready process to run.

Execution Mode

An operating system includes the following components:

- kernel

- Shell

- Utilities

A computer provides two *execution modes* to execute programs.

- *kernel mode*, also known as the *supervisor mode*, the *system mode*, or the *privileged mode*.

- Other programs are executed as a process in the *user mode*. Such a process is - called a *user process*.

The execution mode can be switched back and forth between user and kernel modes.

Process Context

The *context* of a process is that part of the program state that is stored in the processor registers.

A *context switch* is the action to save the current process context and to load a new context.

Parallelism Issues

Parallel programming is more complex than sequential programming. Many additional issues arise.

Homogeneity in Processes

This refers to the similarity of component processes in a parallel program. There are three basic possibilities:

SPMD: The component processes in a ***single-program-multiple-data*** program are ***homogeneous***, in that the same code is executed by multiple processes on different data domains.

MPMD: The component processes in a ***multiple-program-multiple-data*** program are ***heterogeneous***, in that multiple processes may execute different codes.

SIMD: Multiple processes execute the same code and must all execute the same instruction at the same time. In other words, SIMD programs is a special case of SPMD programs.

A *data-parallel* program refers to an SPMD program in general and a program that uses only the data-parallel constructs (such as those in Fortran 90) in particular.

A *functional-parallel* (also known as *task-parallel* or *control-parallel*) program is usually a synonym for an MPMD program.

Parallel Block

A natural way to express MPMD programs is to use the *parbegin* and *parend* constructs.

parbegin S_1 S_2 **...** S_n **parend**

is called a *parallel block*, where S_1 S_2 **...** S_n **are its component processes, which could contain different codes.**

Parallel Loop

When all processes in a parallel block share the same code, we can denote the parallel block with a shorthand notation called a *parallel loop* as follows:

```
parbegin Process(1) . . . Process(n) parend
```

can be simplified to the following parallel loop :

```
parfor ( i=1; i<=n; i++) { Process(i) }
```

Heterogeneous Processes

When the number of different codes is small, one can fake MPMD by using an SPMD program. For instance, the MPMD code parbegin A; B; C; parend can be expressed as an SPMD parallel loop

```
    parfor (i=0; i<3; i++) {
   if (i==0) A;
      if (i==1) B;
      if (i==2) C;
}
```

Multi-Code versus Single-Code

MPPs and COWs use MPMD using the *multi-code* approach.

```
run A on node 0
run B on node 1
run C on node 2
```

- An SPMD program can be specified using the *single-code* approach. For instance, to specify the parallel loop

```
parfor (i = 0; i < N; i++) { foo(i) }
```

the user needs to write only one program such as the following:

```
pid = my_process_id();
numproc = number_of_processes() ;
for ( i = pid; i < N; i = i + numproc) foo(i) ;
```

Data-Parallel Constructs

In data-parallel languages, SPMD parallelism can be specified using data parallel constructs. For instance, to specify the parallel loop:

```
parfor (i = 1; i <= N; i++) {C[i]=A[i]+B[i];}
```

the user can use an array assignment: C = A + B or the following loop:

```
forall (i = 1, N) C[i]=A[i]+B[i].
```

Static versus Dynamic Parallelism

A program exhibits *static parallelism* if its structure and the number of component processes can be determined before run time (e.g., at compile time, link time, or load time).

parbegin *P*, *Q*, *R* parend is static if *P*, *Q*, *R* are.

***Dynamic parallelism* implies that processes can be created and terminated at run time.**

while (C>0) begin fork(foo(C)); C=boo(C); end

Static parallelism can be expressed by the constructs such as parallel blocks and parallel loops. Dynamic parallelism is usually expressed through some kind of *fork* and *join* operations.

Fork/Join

There have been different versions of fork/join. Example:

```
Process A:
begin
  Z = 1;
  fork(B);
  T = foo(3);
end
```

```
Process B:
begin
    fork(C);
    X = foo(Z);
    join(C);
    output(X+Y);
end
```

```
Process C:
begin
    Y = boo(Z);
end
```

Join statement is used to make the parent wait for the child.

Fork and join are very flexible constructs.

Parallelism

Degree of parallelism (DOP) of a parallel program is usually defined as the number of component processes that can be executed simultaneously.

Granularity is defined to be the computation workload executed between two parallelism or interaction operations.

The *unit of granularity* is number of instructions, number of floating-point operations, or seconds.

A rough classification is to consider grain size to be fine (small) for less than 200, medium for 20 to 2000, and large (coarse) for thousands or more computation operations.

operation-level (or *instruction-level*) *parallelism* when the component processes in a parallel program are as small as just one or a few computation operations or instructions.

The term *block-level* refers to the case when the size of individual processes is a block of assignment statements.

A special case of block-level parallelism is ***loop-level***, where a loop creates a number of processes, each executing an iteration consisting of a number of assignment statements.

When the component processes each consist of a procedure call, we have ***procedure-level*** parallelism, sometimes also called ***task-level*** parallelism.

The degree of parallelism and the grain size are often reciprocal

The degree of parallelism and the overhead of communication and synchronization usually have a proportional relationship.

With **explicit allocation**, the user needs to explicitly specify how to allocate data and workload.

With ***implicit allocation***, this task is done by the compiler and the runtime system. Various combinations are possible.

In distributed memory systems, a popular method is the ***owner-compute rule***: how to allocate data so that most of time, data needed by a process are nearby.

Point to Point Communication

Communication through shared variable:

In a shared-memory program, one process can compute a value of a shared variable and store it in the shared memory. Later on another process can get this value by referencing the variable.

Communication through parameter passing:

Data values can be passed as parameters between the child process and the parent.

Communicate through message passing:

In a multicomputer model, processes exchange data through send and receive.

Message Passing /Communication Issues

An *interaction or communication* is an activity or operation that processes perform to affect the behavior of one another.

Three most frequently used types of interactions:

- synchronization

- Point to point communication

- Aggregation (collective)

Synchronization

A *synchronization* operation causes processes to wait for one another, or allows processes that are waiting to resume execution.

Aggregation

An ***aggregation*** **operation is a set of steps used to merge partial results computed by the component processes of a parallel program to generate a complete result.**

Consider the inner product of two vectors A and B.

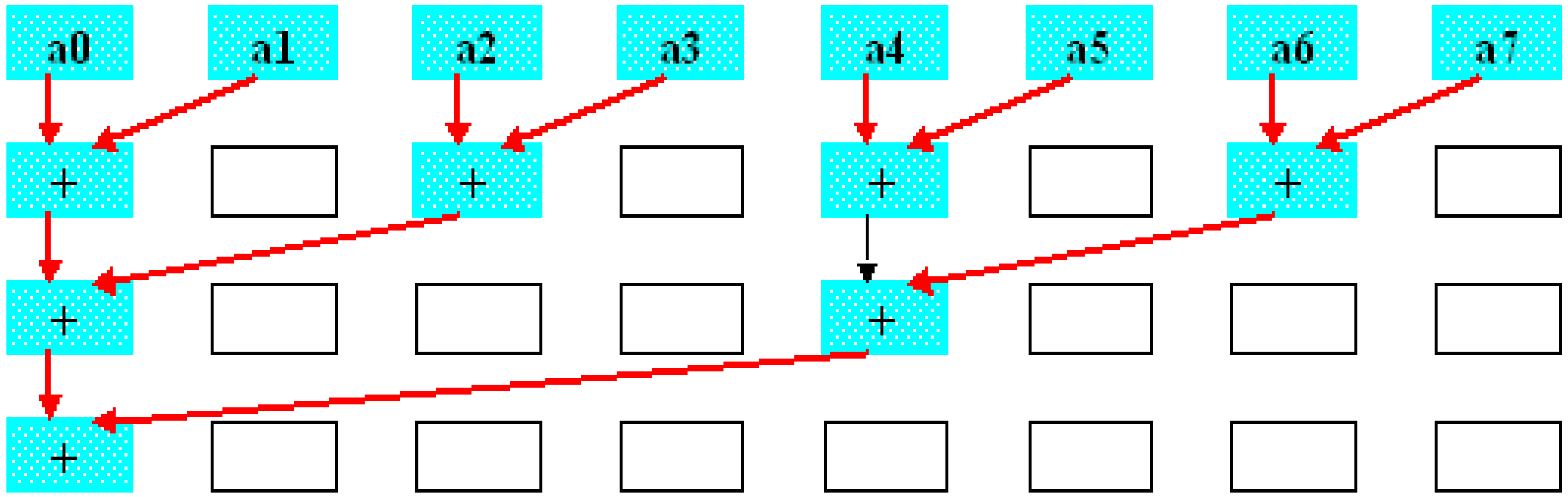
```
parfor (i=0; i <n; i++)
  X[i]= A[i] * B[i];

  X[i] = X[i] + X[i-1];

barrier();

inner_product = aggregate_sum(X[i]);
```

This summation operation is called a *reduction*

A recursive doubling reduction operation

Suppose there are *n* processes *P*(0), *P*(1),..., P(*n*–1). An array element a[i] is initially distributed to process *P*(*i*).

```
Sum = a[i];                           // Each process has a local variable Sum
for (j=1; j<n; j=j*2) {               // there are log(n) supersteps
    if ( i % j == 0) {
        get Sum of process P(i+j) into a local variable tmp;
        Sum = Sum + tmp;
    }
}
```



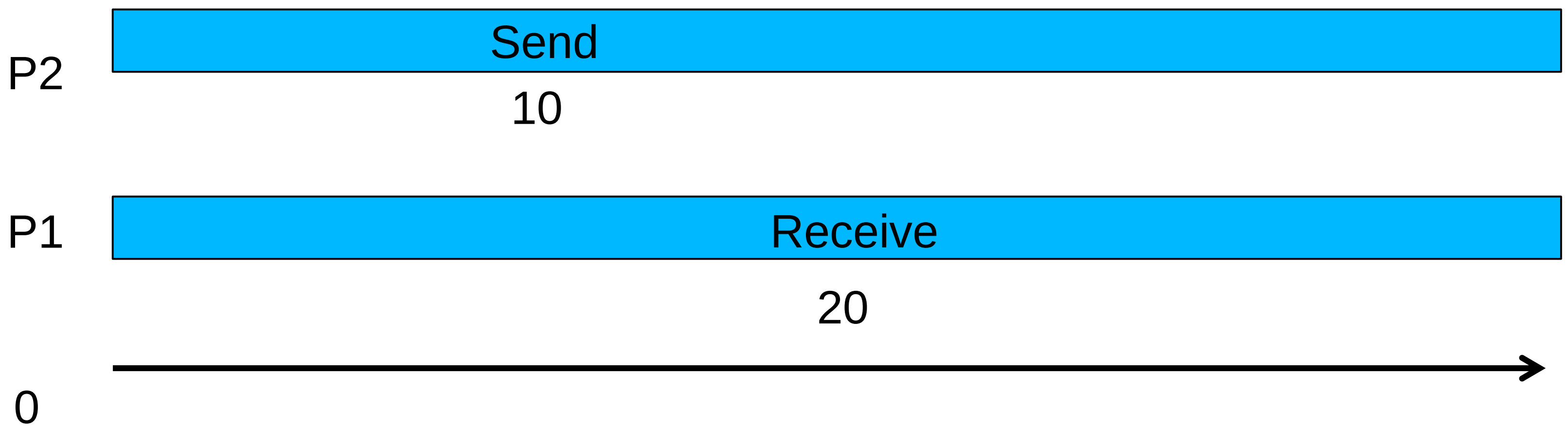
Message-Passing Modes

A two-party communication is called a *point-to-point* communication, when one process sends a message to another process. A multiparty communication is often called a *collective* communication. Either type of communication can be:

Synchronous: All participants must arrive before the interaction begins. A simple way to tell if an interaction is synchronous is the *two-barrier test*:

Blocking: A participant can enter an interaction as soon as it arrives, regardless of where the other parties are. It can exit the interaction when it has finished its portion of the interaction. Other parties may have not finished (or even entered) the interaction code.

Nonblocking: A participant can enter an interaction as soon as it arrives. It can exit the interaction before it has finished its portion of the interaction. An example is a nonblocking send, whose completion implies only that the process has requested to send. The message is not necessarily sent out.


P2
Send
10
P1
Receive
20
0

Communication Patterns

The *pattern* of an interaction refers to which participants affect which other participants.

***The* pattern is static if can be determined at compile time. Otherwise it is a *dynamic*.**

In an *n*-party interaction, if the pattern can be specified as a simple function of the process index, we say the interaction has a *regular* pattern.

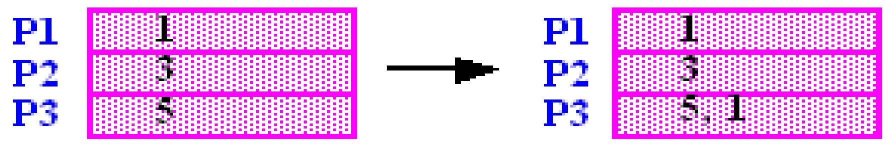
- One-to-One

- One-to-Many

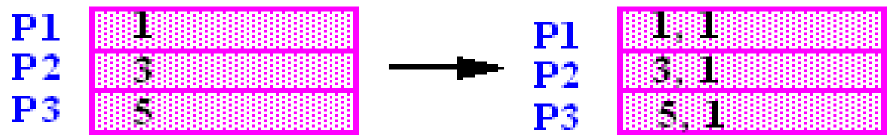
- Many-to-One

- Many-to-Many

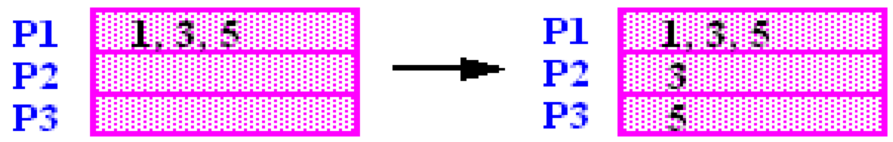
Regular or irregular communications can be modelled by the *h-relation*.



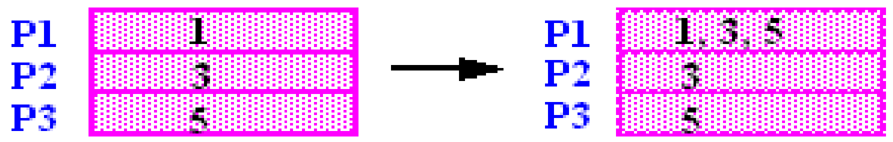
(a) Point-to-point : P1 sends 1 to P3



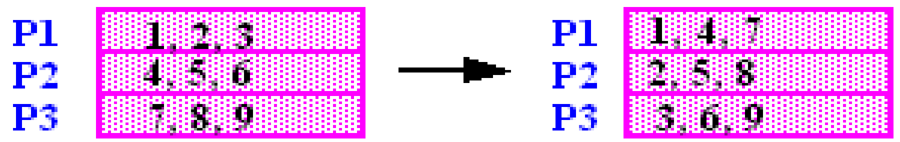
(b) Broadcast : P1 sends 1 to all



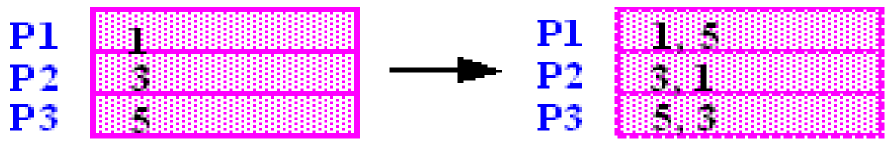
(c) Scatter: P1 sends one value to each node



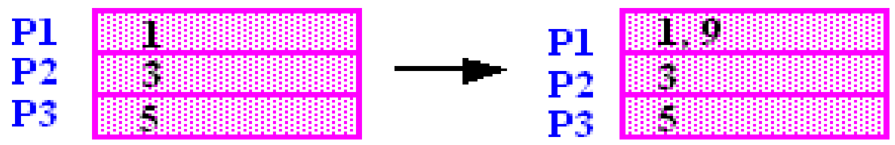
(d) Gather: P1 gets one value from each node



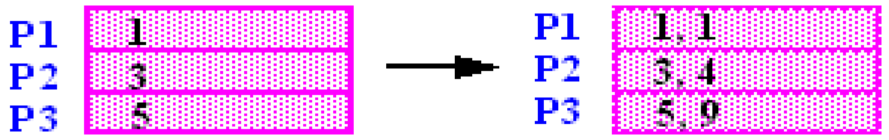
(e) Total exchange: each node sends a distinct message to every node



(f) Shift: each node sends one value to the next and receives one from the previous



(g) Reduction: P1 gets the sum 1 + 3 + 5 = 9



(h) Scan: P1 gets 1, P2 gets 1 + 3 = 4, and P3 gets 1 + 3 + 5 = 9

Point-to-point and collective communication operations

Health and medical Applications on supercomputing

1. **Google**
2. **Google scholar**
3. **Citeceer**
4. **UTA Library**
    1. **IEEE Xplore Database**
    2. **Elsevier database, Science Direct**
    3. **ACM database**
    4. **Springer**

Searching for papers

1. Parallel Medical imaging (Manukonda, Noore, Shrestha, Kambham
2. Parallel AI (Cruel, Tram, Alam, Parne, Piprudiya, Mena
3. Parallel Earth Sciences (Sollner, Jay Simha, Murali, Kasturi, Sungomula,
4. Parallel Energy (Chi,
5. DNA sequencing (Snigdha, Khakurel,
6. Genome Assembly
7. Drug Discovery (Gatlin,
8. Covid (Bahbouh,
9. Genetics (Khan,
10. Brain Simulation (Lueckenhoff,
11. Computational Biology (Deweese,
12. Transportation
13. Data mining (Gade, Thorupunoori, Hedaoo,
14. Human body Simulation (Bonthala,
15. Models of human physiology
16. Heart, lung and other organ Modelling
17. Natural Language Processing (Zhang,
18. Parallel Vision (Deepak, Allola