Parallel & Distributed Computing Techniques

Lâm Ngọc Ẩn
Tăng Trường Tuyến
Huỳnh Tuấn Thông
Đặng Hà Thế Hiển

6/12/2011
Contents

1. Message-Passing Computing
2. Partitioning & Divide-And-Conquer Strategies
3. Synchronous Computation
4. Embarrassingly Parallel Computations
5. Pipelined Computations
6. Load Balancing & Termination Detection
1. Message-Passing Computing

- Programming a message-passing multicomputer can be achieved by:
  - Designing a special parallel programming language
  - Extending the syntax/reserved words of an existing sequential high-level language to handle message-passing.
  - Using an existing sequential high-level language and providing a library of external procedures for message-passing.
1. Message-Passing Computing

- Message-passing programming using user-level message-passing libraries needs two mechanisms:
  - A method of creating separate processes for execution on different computers
  - A method of sending and receiving messages.
1. Message-Passing Computing

- **Static process creation:**
  - All processes are specified before execution.
  - The system will execute a fixed number of processes.

- **Dynamic process creation**
  - Process can be created and their execution initiated during execution of other processes.
  - Number of processes may vary during execution.
1. Message-Passing Computing

- Static process creation: MPMD model

```
Source file

Compile to suit processor

Executable

Processor 0

Source file

Processor p - 1
```
1. Message-Passing Computing

- **Static process creation: SPMD model**
  - Different processes merged into one program. Control statements select different parts for each processor to execute. All executables started together.
1. Message-Passing Computing

- **Dynamic process creation: MPMD model**
  - Separate programs for each processor. One processor executes master process. Other processes started from within master process.

![Diagram](image_url)

- `spawn();`
1. Message-Passing Computing

- Basic “point-to-point” Send and Receive Routines:
  - Passing a message between processes using `send()` and `recv()` library calls:

```c
send(&x, 2);
recv(&y, 1);
```

Generic syntax (actual formats later)
1. Message-Passing Computing

- **Synchronous Message Passing:**
  - Routines that actually return when message transfer completed.

- **Synchronous send routine:**
  - Waits until complete message can be accepted by the receiving process before sending the message.

- **Synchronous receive routine:**
  - Waits until the message it is expecting arrives.

- Synchronous routines intrinsically perform two actions: transfer data and synchronize processes.
1. Message-Passing Computing

- **Synchronous Message Passing:**

  ![Diagram of synchronous message passing between Process 1 and Process 2.](image)

  - Time
  - Suspend process
  - Both processes continue
  - Send: `send();`
  - Request to send: `send();`
  - Acknowledgment: `recv();`
  - Message: `recv();`
  - Both processes continue

---

6/12/2011
1. Message-Passing Computing

- MPI Definitions of Blocking and Non-Blocking:
  - **Blocking** - return after their local actions complete, though the message transfer may not have been completed.
  - **Non-blocking** - return immediately.
  - Assumes that data storage used for transfer not modified by subsequent statements prior to being used for transfer, and it is left to the programmer to ensure this.
1. Message-Passing Computing

- **Message Tag**
- **“Group” message passing routines**
  - Broadcast
  - Gather
  - Scatter
Contents

1. Message-Passing Computing
2. Partitioning & Divide-And-Conquer Strategies
3. Synchronous Computation
4. Embarrassingly Parallel Computations
5. Pipelined Computations
6. Load Balancing & Termination Detection
Partitioning & Divide-And-Conquer Strategies

Partitioning
- Partitioning simply divides the problem into parts.

Divide and Conquer
- Characterized by dividing problem into sub-problems of same form as larger problem. Further divisions into still smaller sub-problems, usually done by recursion.
2. Partitioning & Divide-And-Conquer Strategies

- Partitioning/Divide and Conquer Examples
  - Operations on sequences of number such as simply adding them together
  - Several sorting algorithms can often be partitioned or constructed in a recursive fashion
  - Numerical integration
  - N-body problem
2. Partitioning & Divide-And-Conquer Strategies

- Partitioning a sequence of numbers into parts and adding the parts

\[ x_0 \ldots x_{(n/p)-1} \quad x_{n/p} \ldots x_{(2n/p)-1} \quad \ldots \quad x_{(p-1)n/p} \ldots x_{n-1} \]
2. Partitioning & Divide-And-Conquer Strategies

❖ Tree construction
2. Partitioning & Divide-And-Conquer Strategies

- Dividing a list into parts

Diagram showing a tree structure with nodes labeled from $P_0$ to $P_7$ representing the division of the original list into parts.
2. Partitioning & Divide-And-Conquer Strategies

- Partial summation
2. Partitioning & Divide-And-Conquer Strategies

- Quadtree
2. Partitioning & Divide-And-Conquer Strategies

- Dividing an image

![Diagram of an image divided into nine parts with annotations for first and second divisions.](image-url)
2. Partitioning & Divide-And-Conquer Strategies

❖ Bucket sort

- One “bucket” assigned to hold numbers that fall within each region. Numbers in each bucket sorted using a sequential sorting algorithm.
2. Partitioning & Divide-And-Conquer Strategies

- Parallel version of bucket sort – Simple approach
Further Parallelization

- Partition sequence into $m$ regions, one region for each processor.
- Each processor maintains $p$ “small” buckets and separates numbers in its region into its own small buckets.
- Small buckets then emptied into $p$ final buckets for sorting, which requires each processor to send one small bucket to each of the other processors (bucket $i$ to processor $i$).
Another parallel version of bucket sort

\( p \) processors

Small buckets

Empty small buckets

Large buckets

Sort contents of buckets

Merge lists

\( n/m \) numbers

Unsorted numbers

Sorted numbers
2. Partitioning & Divide-And-Conquer Strategies

“all-to-all” broadcast routine

- "all-to-all" broadcast routine

- Process 0: Send buffer 0 to p-1
- Process p-1: Receive buffer 0 to p-1

Corresponds to one big bucket

Corresponds to set of small buckets

See also next slide
2. Partitioning & Divide-And-Conquer Strategies

- “all-to-all” broadcast routine

- Diagram showing the process of broadcasting from P0 to P3 and vice versa,
  - P0: [A0,0, A0,1, A0,2, A0,3]
  - P1: [A1,0, A1,1, A1,2, A1,3]
  - P2: [A2,0, A2,1, A2,2, A2,3]
  - P3: [A3,0, A3,1, A3,2, A3,3]

- “All-to-all” broadcast routine:
  - P0 sends to P1: [A0,0, A1,0, A2,0, A3,0]
  - P1 sends to P2: [A0,1, A1,1, A2,1, A3,1]
  - P2 sends to P3: [A0,2, A1,2, A2,2, A3,2]
  - P3 sends to P0: [A0,3, A1,3, A2,3, A3,3]
3. Synchronous Computation

- In a (fully) synchronous application, all the processes synchronized at regular points.

- **Barrier**
  - A basic mechanism for synchronizing processes - inserted at the point in each process where it must wait.
  - All processes can continue from this point when all the processes have reached it (or, in some implementations, when a stated number of processes have reached this point).
3. Synchronous Computation

- Processes reaching barrier at different times
3. Synchronous Computation

- In message-passing systems, barriers provided with library routines:

```
P0                  P1                  Pp-1
Barrier();          Barrier();          Barrier();
```

Processes wait until all reach their barrier call.
3. Synchronous Computation

- Barrier Implementation – Counter implementation
  - a linear barrier
3. Synchronous Computation

- **Barrier Implementation - Counter implementation**
  - Good barrier implementations must take into account that a barrier might be used more than once in a process.
  - Might be possible for a process to enter the barrier for a second time before previous processes have left the barrier for the first time.
3. Synchronous Computation

- **Barrier Implementation - Counter implementation**
  - Counter-based barriers often have two phases:
    - A process enters arrival phase and does not leave this phase until all processes have arrived in this phase.
    - Then processes move to departure phase and are released.
3. Synchronous Computation

- Barrier Implementation - Counter implementation

- Master:
  ```
  for (i = 0; i < n; i++)
  recv(Pany);
  for (i = 0; i < n; i++)
  send(Pi);
  /*count slaves as they reach barrier*/
  /* release slaves */
  ```

- Slave processes:
  ```
  send(Pmaster);
  recv(Pmaster);
  ```
3. Synchronous Computation

- Barrier Implementation - Counter implementation

```c
for(i=0; i<n; i++)
    recv(P\text{any});
for(i=0; i<n; i++)
    send(P_i);
```

Master

Slave processes
3. Synchronous Computation

- **Barrier Implementation - Tree implementation**
  - 1st stage: \( P_1 \) sends message to \( P_0 \); (when \( P_1 \) reaches its barrier)
    - \( P_3 \) sends message to \( P_2 \); (when \( P_3 \) reaches its barrier)
    - \( P_5 \) sends message to \( P_4 \); (when \( P_5 \) reaches its barrier)
    - \( P_7 \) sends message to \( P_6 \); (when \( P_7 \) reaches its barrier)
  - 2nd stage: \( P_2 \) sends message to \( P_0 \); (\( P_2 \) & \( P_3 \) reached their barrier)
    - \( P_6 \) sends message to \( P_4 \); (\( P_6 \) & \( P_7 \) reached their barrier)
  - 3rd stage: \( P_4 \) sends message to \( P_0 \); (\( P_4 \), \( P_5 \), \( P_6 \), & \( P_7 \) reached barrier)
    - \( P_0 \) terminates arrival phase; (when \( P_0 \) reaches barrier & received message from \( P_4 \))

Release with a reverse tree construction.
3. Synchronous Computation

- Barrier Implementation - Tree implementation
3. Synchronous Computation

- **Barrier Implementation – Butterfly Barrier**

  1st stage \( P_0 \leftrightarrow P_1, P_2 \leftrightarrow P_3, P_4 \leftrightarrow P_5, P_6 \leftrightarrow P_7 \)

  2nd stage \( P_0 \leftrightarrow P_2, P_1 \leftrightarrow P_3, P_4 \leftrightarrow P_6, P_5 \leftrightarrow P_7 \)

  3rd stage \( P_0 \leftrightarrow P_4, P_1 \leftrightarrow P_5, P_2 \leftrightarrow P_6, P_3 \leftrightarrow P_7 \)
3. Synchronous Computation

**Local Synchronization**

- Suppose a process $P_i$ needs to be synchronized and to exchange data with process $P_{i-1}$ and process $P_{i+1}$ before continuing:

  - Not a perfect three-process barrier because process $P_{i-1}$ will only synchronize with $P_i$ and continue as soon as $P_i$ allows. Similarly, process $P_{i+1}$ only synchronizes with $P_i$. 

```plaintext
Process $P_{i-1}$  Process $P_i$  Process $P_{i+1}$
recv($P_i$);     send($P_{i-1}$);     recv($P_i$);
send($P_i$);     send($P_{i+1}$);     send($P_i$);
recv($P_{i-1}$); recv($P_{i+1}$);     recv($P_i$);
```
3. Synchronous Computation

**Deadlock**

- When a pair of processes each send and receive from each other, deadlock may occur.
- Deadlock will occur if both processes perform the send, using synchronous routines first (or blocking routines without sufficient buffering). This is because neither will return; they will wait for matching receives that are never reached.
3. Synchronous Computation

- **Deadlock – Solution**
  - Arrange for one process to receive first and then send and the other process to send first and then receive.
  - Combined deadlock-free blocking `sendrecv()` routines

**Example**

```
Process P_{i-1}            Process P_i                          Process P_{i+1}

sendrecv(P_i); ←→ sendrecv(P_{i-1});

sendrecv(P_{i+1}); ←→ sendrecv(P_i);
```
3. Synchronous Computation

❖ Synchronized Computations

❖ Can be classified as:

- In fully synchronous, all processes involved in the computation must be synchronized.
- In locally synchronous, processes only need to synchronize with a set of logically nearby processes, not all processes involved in the computation.
3. Synchronous Computation

- Fully Synchronized Computation - Data Parallel Computations
  - Same operation performed on different data elements simultaneously; i.e., in parallel.
  - Particularly convenient because:
    - Ease of programming (essentially only one program).
    - Can scale easily to larger problem sizes.
    - Many numeric and some non-numeric problems can be cast in a data parallel form.
3. Synchronous Computation

- Fully Synchronized Computation - Data Parallel Computations

- To add the same constant to each element of an array:
  
  ```
  for (i = 0; i < n; i++)
      a[i] = a[i] + k;
  ```

- The statement: `a[i] = a[i] + k;`
  could be executed simultaneously by multiple processors, each using a different index `i (0 < i <= n).`
3. Synchronous Computation

- Fully Synchronized Computation - Data Parallel Computations

```
Instruction
a[] = a[] + k;
```

```
Processors

a[0]=a[0]+k;

a[1]=a[1]+k;

a[n-1]=a[n-1]+k;
```

```
a[0]
a[1]
a[n-1]
```
3. Synchronous Computation

- **Fully Synchronized Computation - Data Parallel Computations**

- **forall construct**: special “parallel” construct in parallel programming languages to specify data parallel operations

  ```
  forall (i = 0; i < n; i++) {
    body
  }
  ```

  states that $n$ instances of the statements of the body can be executed simultaneously.
3. Synchronous Computation

- **Fully Synchronized Computation - Data Parallel Computations**
  - To add \( k \) to each element of an array, \( a \), we can write:
    
    ```
    forall (i = 0; i < n; i++)
    a[i] = a[i] + k;
    ```
  - Data parallel technique applied to multiprocessors and multicomputers
    
    ```
    i = myrank;
    a[i] = a[i] + k; /* body */
    barrier(mygroup);
    ```
3. Synchronous Computation

- **Fully Synchronized Computation - Synchronous Iteration**
  - Each iteration composed of several processes that start together at beginning of iteration. Next iteration cannot begin until all processes have finished previous iteration.
3. Synchronous Computation

- Fully Synchronized Computation - Synchronous Iteration

- Using `forall` construct:
  
  ```
  for (j = 0; j < n; j++)             /*for each synch. iteration */
    forall (i = 0; i < N; i++)     /*N procs each using*/
      body(i);                  /* specific value of i */
  ```

- Using message passing barrier:
  
  ```
  for (j = 0; j < n; j++) {            /*for each synchr.iteration */
    i = myrank;                /*find value of i to be used */
    body(i);
    barrier(mygroup);
  } /*N procs each using*/
  ```
3. Synchronous Computation

- Fully Synchronized Computation - Synchronous Iteration
  - Solving a General System of Linear Equations by Iteration

\[
\begin{align*}
    a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 & \ldots + a_{n-1,n-1}x_{n-1} = b_{n-1} \\
    \vdots \quad \vdots \quad \vdots \quad \vdots & \quad \vdots \quad \vdots \quad \vdots \\
    a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 & \ldots + a_{2,n-1}x_{n-1} = b_2 \\
    a_{1,0}x_0 + a_{1,1}x_1 + a_{1,2}x_2 & \ldots + a_{1,n-1}x_{n-1} = b_1 \\
    a_{0,0}x_0 + a_{0,1}x_1 + a_{0,2}x_2 & \ldots + a_{0,n-1}x_{n-1} = b_0
\end{align*}
\]
3. Synchronous Computation

- Locally Synchronized Computation - Heat Distribution Problem

In this example assumes a square boundary
Contents

1. Message-Passing Computing
2. Partitioning & Divide-And-Conquer Strategies
3. Synchronous Computation
4. Embarrassingly Parallel Computations
5. Pipelined Computations
6. Load Balancing & Termination Detection
A computation that can obviously be divided into a number of completely independent parts, each of which can be executed by a separate process(or).

No communication or very little communication between processes
Each process can do its tasks without any interaction with other processes
4. Embarrassingly Parallel Computations

- **static process creation and master-slave approach**

![Diagram showing the process of static process creation and master-slave approach]

- All processes started together
- Send initial data
- Collect results
- Send
- Receive

**Usual MPI approach**
4. Embarrassingly Parallel Computations

- dynamic process creation and master-slave approach

(PVM approach)

Start Master initially

spawn()  
send()  
recv()  

Send initial data

Collect results

recv()  
send()  

Slaves
Mandelbrot Set

Set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limit) when computed by iterating the function

\[ z_{k+1} = z_k^2 + c \]

where \( z_{k+1} \) is the \((k + 1)th\) iteration of the complex number \( z = a + bi \) and \( c \) is a complex number giving position of point in the complex plane. The initial value for \( z \) is zero.

Iterations continued until magnitude of \( z \) is greater than 2 or number of iterations reaches arbitrary limit. Magnitude of \( z \) is the length of the vector given by

\[ z_{\text{length}} = \sqrt{a^2 + b^2} \]
Sequential routine computing value of one point returning number of iterations

```c
structure complex {
    float real;
    float imag;
};
int cal_pixel(complex c)
{
    int count, max;
    complex z;
    float temp, lengthsq;
    max = 256;
    z.real = 0; z.imag = 0;
    count = 0; /* number of iterations */
    do {
        temp = z.real * z.real - z.imag * z.imag + c.real;
        z.imag = 2 * z.real * z.imag + c.imag;
        z.real = temp;
        lengthsq = z.real * z.real + z.imag * z.imag;
        count++;
    } while (lengthsq < 4.0) && (count < max));
    return count;
}
```
Mandelbrot set
Parallelizing Mandelbrot Set Computation

**Static Task Assignment**

Simply divide the region in to fixed number of parts, each computed by a separate processor.

Not very successful because different regions require different numbers of iterations and time.

**Dynamic Task Assignment**

Have processor request regions after computing previous regions
Contents

1. Message-Passing Computing
2. Partitioning & Divide-And-Conquer Strategies
3. Synchronous Computation
4. Embarrassingly Parallel Computations
5. Pipelined Computations
6. Load Balancing & Termination Detection
5. Pipelined Computations

- Problem divided into a series of tasks that have to be completed one after the other (the basis of sequential programming). Each task executed by a separate process or processor.
5. Pipelined Computations

1. If more than one instance of the complete problem is to be executed
2. If a series of data items must be processed, each requiring multiple operations
3. If information to start next process can be passed forward before process has completed all its internal operations
“Type 1” Pipeline Space-Time Diagram
“Type 2” Pipeline Space-Time Diagram

(a) Pipeline structure

(b) Timing diagram

Input sequence
\[ d_9d_8d_7d_6d_5d_4d_3d_2d_1d_0 \]

\[ P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5 \rightarrow P_6 \rightarrow P_7 \rightarrow P_8 \rightarrow P_9 \]
“Type 3” Pipeline Space-Time Diagram

Information transfer sufficient to start next process

Information passed to next stage

(a) Processes with the same execution time

(b) Processes not with the same execution time
Solving a System of Linear Equations

Upper-triangular form

\[
a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 + \ldots + a_{n-1,n-1}x_{n-1} = b_{n-1}
\]

\[
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2
\]

\[
a_{1,0}x_0 + a_{1,1}x_1 = b_1
\]

\[
a_{0,0}x_0 = b_0
\]

where \( a \)'s and \( b \)'s are constants and \( x \)'s are unknowns to be found.
Back Substitution

First, unknown $x_0$ is found from last equation; i.e.,

\[ x_0 = \frac{b_0}{a_{0,0}} \]

Value obtained for $x_0$ substituted into next equation to obtain $x_1$; i.e.,

\[ x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}} \]

Values obtained for $x_1$ and $x_0$ substituted into next equation to obtain $x_2$:

\[ x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}} \]

and so on until all the unknowns are found.
Pipeline Solution

First pipeline stage computes $x_0$ and passes $x_0$ onto the second stage, which computes $x_1$ from $x_0$ and passes both $x_0$ and $x_1$ onto the next stage, which computes $x_2$ from $x_0$ and $x_1$, and so on.

Type 3 pipeline computation
The \( i \)th process \((0 < i < n)\) receives the values \( x_0, x_1, x_2, \ldots, x_{i-1} \) and computes \( x_i \) from the equation:

\[
x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j} x_j}{a_{i,i}}
\]
Sequential Code

Given constants \( a_{i,j} \) and \( b_k \) stored in arrays \( a[ ][ ] \) and \( b[ ] \), respectively, and values for unknowns to be stored in array, \( x[ ] \), sequential code could be

\[
\begin{align*}
x[0] &= \frac{b[0]}{a[0][0]}; & \text{computed separately} \\
\text{for (i = 1; i < n; i++) } \{ & \text{for remaining unknowns*} \\
& \text{sum = 0;} \\
& \text{for (j = 0; j < i; j++) } \\
& \quad \text{sum = sum + a[i][j]*x[j];} \\
& \quad x[i] = \frac{b[i] - \text{sum}}{a[i][i]}; \\
\}
\end{align*}
\]
Parallel Code:

The pseudo code of process $P_i$ ($1 < i < n$) of the pipelined version could be:

```c
for (j = 0; j < i; j++) {
    recv(P i-1, x[j]); // Receive $x_0, x_1,..$ from $P(i-1)$
    send(P i+1, x[j]); // Send $x_0, x_1,..$ from $P(i-1)$
    sum = sum + a[i][j]*x[j]; // Compute sum term
}
sum = 0;
x[i] = (b[i] - sum)/a[i][i]; // Compute $x_i$
send(Pi+1, x[j]); // Send $x_i$ to $P(i+1)$
```
Contents

1 Message-Passing Computing
2 Partitioning & Divide-And-Conquer Strategies
3 Synchronous Computation
4 Embarrassingly Parallel Computations
5 Pipelined Computations
6 Load Balancing & Termination Detection
6. Load Balancing & Termination Detection

- **Load balancing** – used to distribute computations fairly across processors in order to obtain the highest possible execution speed.

- **Termination detection** – detecting when a computation has been completed. More difficult when the computation is distributed.
Load Balancing

(a) Imperfect load balancing leading to increased execution time

(b) Perfect load balancing
Static Load Balancing

- **Round robin algorithm** — passes out tasks in sequential order of processes coming back to the first when all processes have been given a task
- **Randomized algorithms** — selects processes at random to take tasks
- **Recursive bisection** — recursively divides the problem into sub-problems of equal computational effort while minimizing message passing
- **Simulated annealing** — an optimization technique
- **Genetic algorithm** — another optimization technique
Dynamic Load Balancing

- Centralized dynamic load balancing
- Decentralized dynamic load balancing
Centralized dynamic load balancing

- **Advantage:** The master process terminates the computation when
  - The task queue is empty, and
  - Every process has made a request for more tasks without any new tasks been generated.

- **Disadvantages:**
  - High task queue management overheads/load on master process.
  - Contention over access to single queue may lead to excessive contention delays.
Tasks could be transferred by one of two methods:
- Receiver-initiated method.
- Sender-initiated method.
Fully Distributed Work Pool

Requests/tasks

Process

Process

Process

Process
Termination Detection for Decentralized Dynamic Load Balancing

Message passing
Termination Detection for Decentralized Dynamic Load Balancing

Ring termination detection algorithm

Token passed to next processor when reached local termination condition
Program Example: Shortest Path Algorithm

(a) Adjacency matrix
Stages in Searching a Graph

After examining $A$ to $B$:

After examining $B$ to $F$, $E$, $D$, and $C$:

After examining $D$ to $E$:
Sequential Code:

```c
while ((i=next_vertex())!=no_vertex)
    while (j=next_edge(vertex)!=no_edge)
        newdist_j=dist[i] + w[i][j];
        if (newdist_j < dist[j]) {
            dist[j]=newdist_j;
            append_queue(j); }
```
Master
recv(any, P_i); /* request for task from process P_i */
if ((i = next_edge() != no_edge)
    send(P_i, i, dist[i]); /* send next vertex, and */
        /* current distance to vertex */
recv(P_j, j, dist[j]); /* receive new distances */
append_queue(j); /* append vertex to queue */
Parallel Implementation using Centralized Work Pool

Slave (process i)

send($P_{\text{master}}$, $P_i$); /* send a request for task */
recv($P_{\text{master}}$, i, d); /* get vertex number and distance */

while (j=next_edge(vertex) != no_edge) {
  /* get next link around vertex */
  newdist_j = d + w[i][j];
  if (newdist_j < dist[j]) {
    dist[j] = newdist_j;
    send($P_{\text{master}}$, j, dist[j]); /* send back updated distance */
  }
}

} /* no more vertices to consider */

Done
Parallel Implementation Using Decentralized Work Pool
Parallel Implementation
Using Decentralized Work Pool

Master
if ((i = next_vertex()) != no_vertex)
    send(P\textsubscript{i}, "start"); /* start up slave process i */

Slave (process i)
if (recv(P\textsubscript{j}, msgtag = 1)) /* asking for distance */
    send(P\textsubscript{j}, msgtag = 2, dist[i]); /* sending current distance */
if (nrecv(P\textsubscript{master}) { /* if start-up message */
    while (j = next_edge(vertex) != no_edge) { /* get next link around vertex */
        newdist\textsubscript{j} = dist[i] + w[j];
        send(P\textsubscript{j}, msgtag=1); /* Give me the distance */
        recv(P\textsubscript{i}, msgtag = 2, dist[j]); /* Thank you */
        if (newdist\textsubscript{j} > dist[j]) {
            dist[j] = newdist\textsubscript{j};
            send(P\textsubscript{j}, msgtag=3, dist[j]); /* send updated distance to proc. j */
        }
    }
}
References


Thank You!