

Modelo de computación BSP

Programación paralela y distribuida

Fernando Pérez Costoya – Noviembre de 2014

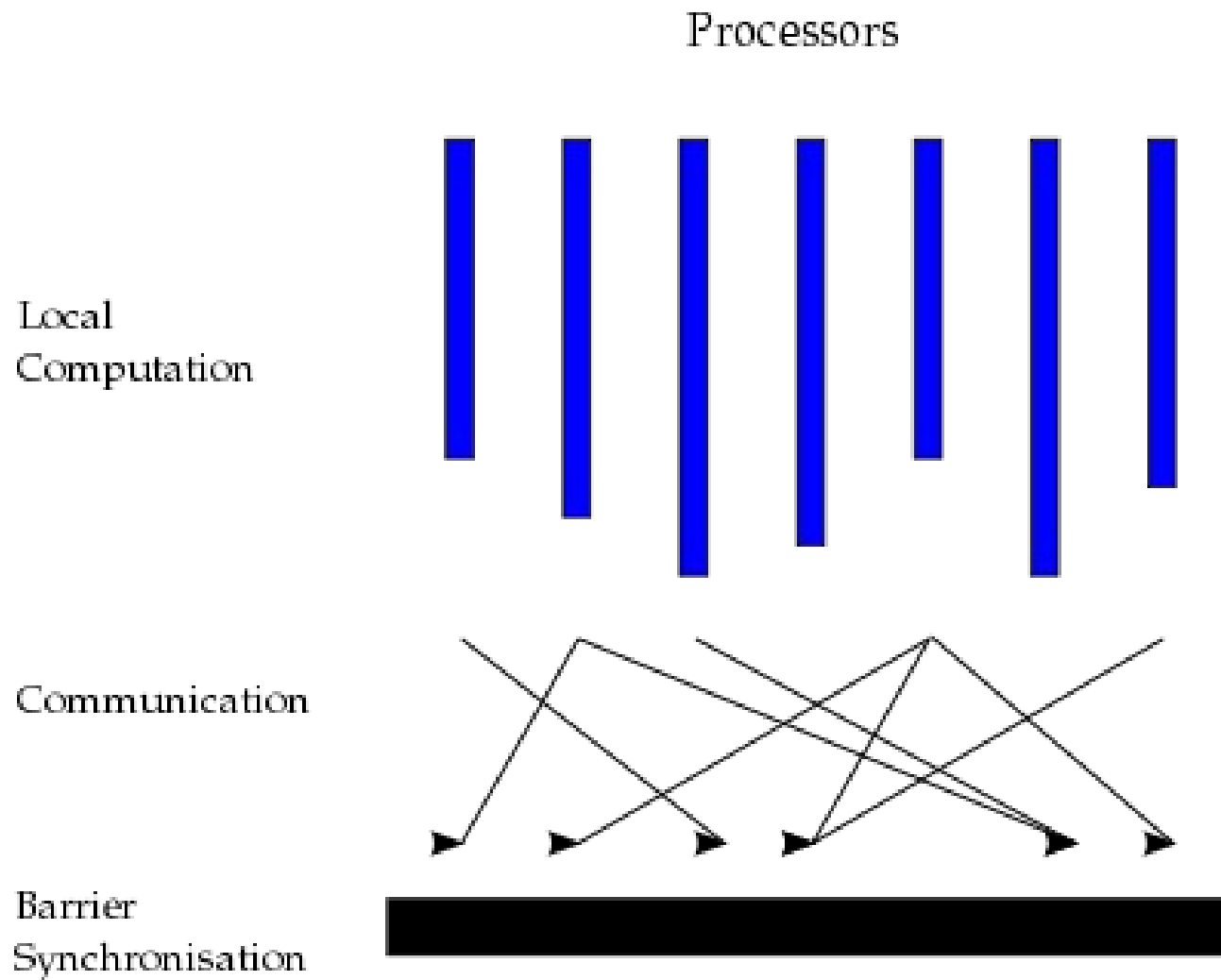
Bulk Synchronous Parallel (Valiant1990)

- Modelo de computación planteado como la
 - Arquitectura von Neumann para sistemas paralelos
 - Puente entre diseñadores HW y SW
 - HW diseñado para que dé soporte al modelo
 - Adecuado para sistemas con memoria compartida y distribuida
 - SW diseñado para que ejecute sobre el modelo
- Impacto limitado; retomado actual. (Apache Hama)
- Elementos del modelo:
 - Componentes con capacidad de procesamiento
 - Componente de comunicación (*router*)
 - Componente de sincronización (tipo barrera)

Modelo de ejecución de BSP

- Síncrono organizado como secuencia *supersteps*
- En *superstep* un componente de procesamiento:
 - Recibe mensajes de otros componentes
 - Realiza procesamiento local
 - Puede enviar mensajes a otros componentes proces.
 - Sincronización tipo barrera al final del *superstep*

Ciclo ejecución BSP (wikipedia)



Modelos programación para grafos

Pregel

Programación paralela y distribuida

Fernando Pérez Costoya – Noviembre de 2014

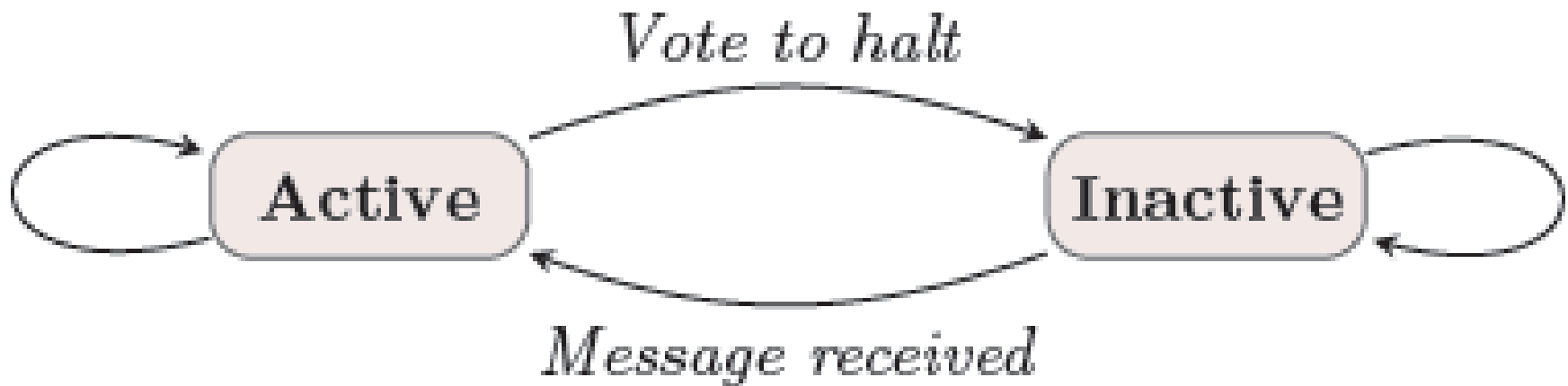
Procesamiento de grafos

- Necesidad de procesar grafos de gran escala
 - Internet, redes sociales, rutas, etc.
 - Aplicación de algoritmos clásicos de grafos
 - Requieren sistemas paralelos/distribuidos
- ¿Modelo programación/*framework*/implementación que oculte sincro, comun, planificación, tolerancia..?
 - ¿MapReduce?
 - Programación no intuitiva
 - Ineficiencia: grafo viaja continuamente entre los nodos
 - Propuesta de Google: Pregel (2010)
 - Basado en BSP; Apache Giraph: versión de libre distribución

Pregel: modelo de programación

- Grafo dirigido tal que cada vértice tiene:
 - ID único; Valor asociado modificable
 - Conjunto de aristas salientes
 - Cada arista: valor asociado + vértice destino
 - Una función asociada (la misma para todos)
- Ejecución de un trabajo Pregel
 - Entrada \rightarrow Grafo; Salida \rightarrow Grafo (con la solución)
 - Secuencia de *supersteps*
 - Inicialmente todos los vértices activos
 - Fin de computación cuando todos los vértices inactivos
 - Recepción de mensaje reactiva vértice

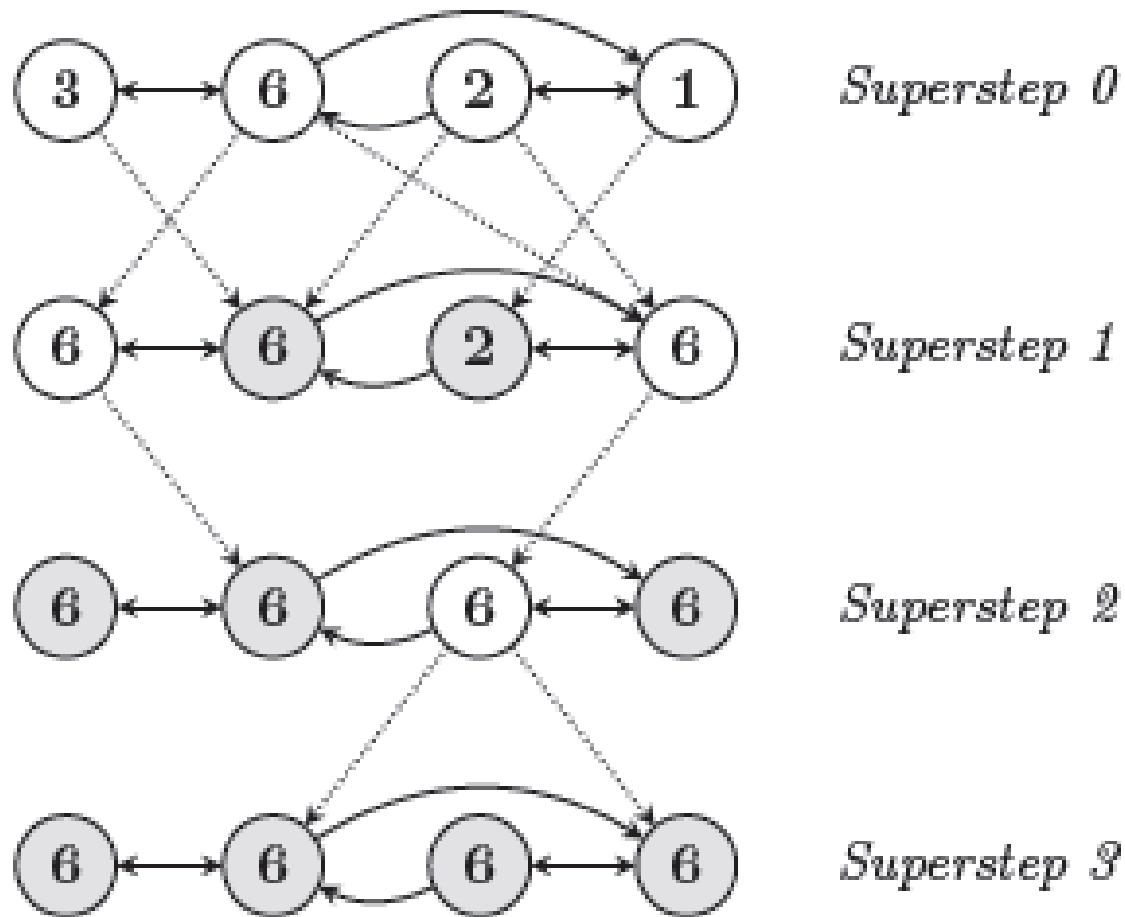
Evolución del estado de un vértice



Ejecución de un *superstep*

- Ejecuta en paralelo la función sobre cada vértice
 - Esa función implementa la lógica del algoritmo
- La ejecución de la función en un vértice V puede:
 - Recibir mensajes enviados a V en *superstep* previo
 - Modificar valor del vértice y de las aristas salientes
 - Enviar mensajes a otros vértices
 - Conectados a V o a cualquiera si conoce su ID
 - Los mensajes se recibirán en siguiente *superstep*
 - Modificar la topología del grafo
 - Añadir/eliminar vértices y aristas

Ejemplo de cálculo del máximo



API en C++

- *Compute*: función de vértices
- *GetValue|MutableValue*: obtiene/cambia valor vértice
- *GetOutEdgeIterator*: inspecciona/cambia aristas sal.
- *SendMessageTo*: envío fiable pero no garantía orden
- *VoteToHalt*: nodo se hace inactivo
- Componentes adicionales:
 - *Combiners*: combinan mensajes destinados a mismo V
 - *Aggregators*: cada vértice puede provee valor a *aggreg.*
 - Sistema agrega valores y los deja disponibles en sig. superstep
- Mutaciones pueden causar conflictos
- E/S: formatos y soportes configurables

Clase Vértice

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
public:
    virtual void Compute(MessageIterator* msgs) = 0;

    const string& vertex_id() const;
    int64 superstep() const;

    const VertexValue& GetValue();
    VertexValue* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();

    void SendMessageTo(const string& dest_vertex,
                      const MessageValue& message);
    void VoteToHalt();
};
```

Implementación

- Maestro-Trabajadores; Pasos ejecución trabajo:
 - M determina particiones en grafo y las asigna a T
 - Por defecto, $hash(ID \text{ vértice})$; puede redefinirlo la aplicación
 - M determina particiones en entrada y asigna a T
 - T lee partición; se queda con sus vértices y reenvía otros
 - Fin de la entrada \rightarrow todos los vértices activos
 - M indica a todo T inicio *superstep*
 - T ejecuta función para cada uno de sus vértices
 - Entregando a cada vértice mensajes recibidos en *superstep* previo
 - Recoge mensajes enviados y transmite empaquetados asíncronamente
 - T señala a M fin de *superstep* especificando n° vértices activos
 - M espera respuesta de todos los T (barrera)
 - Si n° total activos=0 \rightarrow Fin: M solicita a T volcado de resultados

Tolerancia a fallos

- *Checkpoint* (CHK) en almacenamiento persistente
 - Valores de vértices, aristas y mensajes entrantes
- Al inicio de un *superstep* **M** solicita CHK a los **I**
- Caída de **I**
 - Reasigna particiones de **I** a otros trabajadores
 - **M** solicita a todos los **I** que recuperen último CHK
 - Y vuelvan a ejecutar desde ese *superstep*
 - Trabajo repetido por parte de trabajadores que no se han caído
- Recuperación confinada bajo desarrollo (en 2010)
 - CHK de mensajes salientes
 - Reduce gasto innecesario recursos: sólo nodos caídos

Aplicaciones: *PageRank*

```
class PageRankVertex
    : public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *MutableValue() =
                0.15 / NumVertices() + 0.85 * sum;
        }

        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
    }
};
```

Aplicaciones: *Shortest Path*

```
class ShortestPathVertex
: public Vertex<int, int, int> {
void Compute(MessageIterator* msgs) {
    int mindist = IsSource(vertex_id()) ? 0 : INF;
    for (; !msgs->Done(); msgs->Next())
        mindist = min(mindist, msgs->Value());
    if (mindist < GetValue()) {
        *MutableValue() = mindist;
        OutEdgeIterator iter = GetOutEdgeIterator();
        for (; !iter.Done(); iter.Next())
            SendMessageTo(iter.Target(),
                           mindist + iter.GetValue());
    }
    VoteToHalt();
}
};
```


Combiner para Shortest Path

```
class MinIntCombiner : public Combiner<int> {  
    virtual void Combine(MessageIterator* msgs) {  
        int mindist = INF;  
        for (; !msgs->Done(); msgs->Next())  
            mindist = min(mindist, msgs->Value());  
        Output("combined_source", mindist);  
    }  
};
```