# Documento Técnico: Estrategia de Paralelismo y Computación de Alto Rendimiento para PIGEM

## 1. Introducción y Contexto

La **Plataforma Integral de Gestión de Evacuaciones Multiamenaza (PIGEM)** enfrenta desafíos computacionales extraordinarios. Simular evacuaciones masivas en tiempo real para poblaciones de cientos de miles de personas requiere capacidades de cómputo que exceden ampliamente las posibilidades de arquitecturas tradicionales. Este documento presenta la estrategia integral de **Computación de Alto Rendimiento (HPC)** diseñada específicamente para PIGEM.

### 1.1 Magnitud del Desafío Computacional

**Tabla 1: Escala de Simulaciones Objetivo en PIGEM**

| **Escenario** | **Población** | **Agentes** | **Área (km²)** | **Duración Sim.** | **Cómputo Requerido** | **Tiempo Objetivo** |
| --- | --- | --- | --- | --- | --- | --- |
| **Barrio Local** | 5,000 | 5,000 | 1 | 30 min | 10⁹ operaciones | <2 minutos |
| **Comuna** | 50,000 | 50,000 | 25 | 1 hora | 10¹¹ operaciones | <10 minutos |
| **Ciudad Media** | 200,000 | 200,000 | 100 | 2 horas | 10¹² operaciones | <30 minutos |
| **Región Metropolitana** | 1,000,000 | 1,000,000 | 500 | 4 horas | 10¹³ operaciones | <2 horas |
| **Múltiples Ciudades** | 2,000,000 | 2,000,000 | 1,000 | 6 horas | 10¹⁴ operaciones | <4 horas |

### 1.2 Análisis de Complejidad Computacional

**Tabla 2: Complejidad Algorítmica por Componente**

| **Componente** | **Complejidad Temporal** | **Complejidad Espacial** | **Paralelización** | **Escalabilidad** |
| --- | --- | --- | --- | --- |
| **Social Force Model** | O(n²) por timestep | O(n) | Excelente (GPU) | Cuadrática |
| **Pathfinding A**\* | O(b^d) por agente | O(b^d) | Buena (CPU) | Exponencial |
| **Collision Detection** | O(n log n) | O(n) | Excelente (GPU) | Logarítmica |
| **Event Scheduling** | O(log n) por evento | O(n) | Limitada | Logarítmica |
| **Spatial Indexing** | O(n log n) | O(n) | Buena (CPU) | Logarítmica |
| **Visualization** | O(n) por frame | O(n) | Excelente (GPU) | Lineal |

El análisis revela que el **Social Force Model** representa el cuello de botella principal con complejidad O(n²), donde cada agente debe calcular fuerzas de repulsión con todos los agentes vecinos. Esta característica hace que la paralelización masiva sea no solo deseable, sino absolutamente esencial para la viabilidad del sistema.

### 1.3 Arquitectura HPC Híbrida Propuesta

┌─────────────────────────────────────────────────────────────────────────────┐  
│ ARQUITECTURA HPC HÍBRIDA │  
├─────────────────────────────────────────────────────────────────────────────┤  
│ │  
│ ┌─────────────────┐ ┌─────────────────┐ ┌─────────────────────────────┐ │  
│ │ CLUSTER CPU │ │ CLUSTER GPU │ │ CLOUD BURSTING │ │  
│ │ (Lógica Compleja)│ │ (Cálculo Masivo)│ │ (Escalado Dinámico) │ │  
│ │ │ │ │ │ │ │  
│ │ • 64 nodos │ │ • 16 nodos GPU │ │ • AWS/Azure/GCP │ │  
│ │ • 2048 cores │ │ • 128 GPUs │ │ • Kubernetes │ │  
│ │ • 4TB RAM │ │ • 163,840 cores │ │ • Auto-scaling │ │  
│ │ • InfiniBand │ │ • 32GB VRAM │ │ • Spot instances │ │  
│ └─────────────────┘ └─────────────────┘ └─────────────────────────────┘ │  
│ │ │ │ │  
│ └──────────────────────┼──────────────────────────┘ │  
│ │ │  
├─────────────────────────────────┼──────────────────────────────────────────┤  
│ CAPA DE ORQUESTACIÓN │  
│ │ │  
│ ┌─────────────────┐ ┌─────────────────┐ ┌─────────────────────────────┐ │  
│ │ SLURM SCHEDULER │ │ MPI COORDINATOR │ │ LOAD BALANCER │ │  
│ │ │ │ │ │ │ │  
│ │ • Job Queue │ │ • Process Sync │ │ • Resource Monitor │ │  
│ │ • Resource Mgmt │ │ • Data Exchange │ │ • Dynamic Allocation │ │  
│ │ • Priority │ │ • Fault Tolerance│ │ • Performance Metrics │ │  
│ └─────────────────┘ └─────────────────┘ └─────────────────────────────┘ │  
│ │ │  
├─────────────────────────────────┼──────────────────────────────────────────┤  
│ CAPA DE ALMACENAMIENTO │  
│ │ │  
│ ┌─────────────────┐ ┌─────────────────┐ ┌─────────────────────────────┐ │  
│ │ PARALLEL FS │ │ DISTRIBUTED DB │ │ MEMORY FABRIC │ │  
│ │ (Lustre/GPFS) │ │ (ScyllaDB) │ │ (Redis Cluster) │ │  
│ │ │ │ │ │ │ │  
│ │ • 100GB/s I/O │ │ • 1M ops/sec │ │ • 10TB distributed cache │ │  
│ │ • 1PB capacity │ │ • Auto-sharding │ │ • Sub-ms latency │ │  
│ │ • RAID 6 │ │ • Replication │ │ • Consistent hashing │ │  
│ └─────────────────┘ └─────────────────┘ └─────────────────────────────┘ │  
└─────────────────────────────────────────────────────────────────────────────┘

*Figura 1: Arquitectura HPC híbrida para PIGEM*

## 1.4 Fundamentación Científica de la Estrategia HPC

### 1.4.1 Descripción del Problema Computacional

La simulación de evacuaciones masivas en tiempo real presenta un desafío computacional multidimensional que trasciende los límites de las arquitecturas de cómputo tradicionales. Este problema se caracteriza por la necesidad de procesar interacciones complejas entre millones de entidades heterogéneas (agentes) en espacios geográficos extensos, manteniendo coherencia temporal y precisión física, mientras se satisfacen restricciones temporales estrictas para la toma de decisiones operacionales.

**Dimensión de Escalabilidad:** La complejidad computacional inherente del Social Force Model O(n²) para n agentes genera un crecimiento exponencial de la carga computacional. Para simulaciones de evacuación metropolitana (n > 1M agentes), esto resulta en >10¹² operaciones de punto flotante por segundo de simulación, excediendo la capacidad de procesadores convencionales por varios órdenes de magnitud.

**Dimensión de Memoria:** El almacenamiento y acceso eficiente a estados de agentes heterogéneos requiere patrones de memoria complejos que generan problemas de localidad espacial y temporal. Con 1M agentes requiriendo ~1KB de estado cada uno, el footprint de memoria alcanza 1GB, excediendo capacidades de cache típicas y generando latencias de acceso prohibitivas para simulación en tiempo real.

**Dimensión de Comunicación:** Los algoritmos de simulación híbrida ABM-DES requieren sincronización frecuente entre componentes distribuidos, generando overhead de comunicación que puede dominar el tiempo total de ejecución. La coordinación de eventos temporales distribuidos introduce dependencias causales que limitan el paralelismo efectivo.

**Dimensión de Heterogeneidad:** La naturaleza híbrida CPU-GPU del sistema requiere estrategias de particionamiento de carga que consideren las características específicas de cada arquitectura. La transferencia de datos entre espacios de memoria CPU-GPU introduce latencias adicionales que pueden negar los beneficios del paralelismo masivo.

### 1.4.2 Hipótesis de Investigación en HPC

**Hipótesis Principal:** La implementación de una arquitectura híbrida CPU-GPU con estrategias de paralelización específicas por componente (Social Force Model en GPU con complejidad reducida O(n log n), pathfinding distribuido en CPU, coordinación DES mediante actor model) permitirá la simulación en tiempo real de evacuaciones masivas (>1M agentes) manteniendo precisión científica (error <5%) y cumpliendo restricciones temporales operacionales (latencia <30 minutos).

**Hipótesis Secundarias:**

1. **Hipótesis de Optimización Algorítmica:** La aplicación de técnicas de reducción de complejidad (spatial hashing, Fast Multipole Method, hierarchical pathfinding) reducirá la complejidad computacional del Social Force Model de O(n²) a O(n log n) manteniendo precisión >95% comparado con implementaciones exactas.
2. **Hipótesis de Escalabilidad Paralela:** La paralelización masiva en arquitecturas GPU modernas (>10,000 cores) proporcionará speedups >100x comparado con implementaciones CPU secuenciales, manteniendo eficiencia paralela >70% hasta 32 dispositivos GPU.
3. **Hipótesis de Memoria Jerárquica:** La implementación de estrategias de gestión de memoria jerárquica (cache coherente, prefetching predictivo, memory pooling) mejorará la utilización de ancho de banda de memoria en >300% comparado con accesos naive.
4. **Hipótesis de Tolerancia a Fallos:** Los mecanismos de checkpoint distribuido y recuperación automática garantizarán continuidad operacional con overhead <10% y tiempo de recuperación <5 minutos ante fallos de hasta 25% de nodos de cómputo.

### 1.4.3 Pregunta de Investigación en HPC

**Pregunta Principal:** ¿Cómo debe diseñarse una arquitectura de computación híbrida CPU-GPU que optimice la distribución de carga computacional entre diferentes tipos de procesadores para maximizar el rendimiento de simulaciones de evacuación masiva, considerando las características específicas de cada componente algorítmico y las restricciones de memoria, comunicación y sincronización inherentes al problema?

**Preguntas Secundarias:**

1. **Particionamiento de Carga:** ¿Cuáles son las estrategias óptimas de particionamiento entre CPU y GPU que minimizan overhead de comunicación mientras maximizan utilización de recursos computacionales heterogéneos?
2. **Optimización de Memoria:** ¿Qué técnicas de gestión de memoria jerárquica son más efectivas para mantener localidad temporal y espacial en algoritmos de simulación con patrones de acceso irregulares y dependientes de datos?
3. **Sincronización Distribuida:** ¿Cómo pueden implementarse protocolos de sincronización temporal para sistemas híbridos ABM-DES que preserven corrección causal minimizando overhead de coordinación entre procesadores distribuidos?
4. **Escalabilidad Arquitectónica:** ¿Cuáles son los límites fundamentales de escalabilidad de arquitecturas híbridas para simulación de evacuaciones, y qué factores (ancho de banda de memoria, latencia de red, overhead de sincronización) constituyen los cuellos de botella principales?

### 1.4.4 Metodología de Investigación en HPC

La investigación en computación de alto rendimiento seguirá una metodología experimental sistemática:

**Análisis Teórico:** Modelado formal de complejidad computacional y comunicacional para diferentes estrategias de paralelización, incluyendo análisis de escalabilidad asintótica y caracterización de cuellos de botella fundamentales.

**Implementación Experimental:** Desarrollo de prototipos de alta fidelidad que implementen diferentes estrategias de paralelización, utilizando frameworks de desarrollo GPU (CUDA, OpenCL) y bibliotecas de comunicación de alto rendimiento (MPI, NCCL).

**Benchmarking Sistemático:** Evaluación empírica de rendimiento utilizando métricas estándar de HPC (speedup, eficiencia, escalabilidad) en configuraciones de hardware representativas, incluyendo análisis de sensibilidad a parámetros arquitectónicos.

**Optimización Iterativa:** Aplicación de técnicas de profiling de rendimiento (NVIDIA Nsight, Intel VTune) para identificar cuellos de botella específicos y optimización dirigida mediante técnicas algorítmicas y de bajo nivel.

**Validación de Escalabilidad:** Evaluación experimental de escalabilidad en configuraciones de hasta 1000 nodos de cómputo utilizando infraestructura de centros de supercómputo nacionales e internacionales.

## 2. Estrategias de Paralelización por Componente

### 2.1 Paralelización del Social Force Model

El **Social Force Model (SFM)** es el núcleo computacional más intensivo de PIGEM. Su paralelización efectiva determina la viabilidad de simular poblaciones masivas.

#### 2.1.1 Implementación GPU-CUDA

**Tabla 3: Estrategia de Paralelización SFM en GPU**

| **Aspecto** | **Implementación** | **Justificación** | **Rendimiento** | **Limitaciones** |
| --- | --- | --- | --- | --- |
| **Thread Mapping** | 1 thread = 1 agente | Paralelismo natural | 2048 threads/SM | Divergencia de threads |
| **Memory Pattern** | Coalesced access | Maximizar ancho de banda | 900 GB/s | Conflictos de banco |
| **Shared Memory** | Neighbor cache | Reducir accesos globales | 164 KB/SM | Capacidad limitada |
| **Atomic Operations** | Position updates | Evitar race conditions | Hardware support | Serialización |
| **Occupancy** | 75% target | Balance recursos/threads | 1536 threads/SM | Registro spillage |

#### 2.1.2 Algoritmo CUDA Optimizado

\_\_global\_\_ void calculateSocialForces(  
 Agent\* agents,   
 int numAgents,  
 float\* spatialGrid,  
 int gridWidth,  
 int gridHeight,  
 float deltaTime  
) {  
 int agentId = blockIdx.x \* blockDim.x + threadIdx.x;  
   
 if (agentId >= numAgents) return;  
   
 // Cargar agente en memoria local  
 Agent agent = agents[agentId];  
 float2 totalForce = {0.0f, 0.0f};  
   
 // Usar shared memory para vecinos  
 \_\_shared\_\_ Agent neighbors[BLOCK\_SIZE];  
   
 // Determinar celdas vecinas en grid espacial  
 int2 gridPos = worldToGrid(agent.position);  
   
 // Iterar sobre celdas vecinas (3x3)  
 for (int dy = -1; dy <= 1; dy++) {  
 for (int dx = -1; dx <= 1; dx++) {  
 int2 neighborCell = {gridPos.x + dx, gridPos.y + dy};  
   
 if (isValidCell(neighborCell, gridWidth, gridHeight)) {  
 // Cargar vecinos de forma cooperativa  
 loadNeighborsCooperative(neighbors, neighborCell, spatialGrid);  
 \_\_syncthreads();  
   
 // Calcular fuerzas con vecinos  
 for (int i = 0; i < BLOCK\_SIZE; i++) {  
 if (neighbors[i].id != -1 && neighbors[i].id != agent.id) {  
 float2 force = calculatePairwiseForce(agent, neighbors[i]);  
 totalForce.x += force.x;  
 totalForce.y += force.y;  
 }  
 }  
 \_\_syncthreads();  
 }  
 }  
 }  
   
 // Actualizar posición usando integración Verlet  
 float2 acceleration = {totalForce.x / agent.mass, totalForce.y / agent.mass};  
 agent.position.x += agent.velocity.x \* deltaTime + 0.5f \* acceleration.x \* deltaTime \* deltaTime;  
 agent.position.y += agent.velocity.y \* deltaTime + 0.5f \* acceleration.y \* deltaTime \* deltaTime;  
   
 // Actualizar velocidad  
 agent.velocity.x += acceleration.x \* deltaTime;  
 agent.velocity.y += acceleration.y \* deltaTime;  
   
 // Escribir resultado  
 agents[agentId] = agent;  
}

#### 2.1.3 Optimizaciones de Memoria

**Tabla 4: Estrategias de Optimización de Memoria GPU**

| **Técnica** | **Implementación** | **Beneficio** | **Overhead** | **Casos de Uso** |
| --- | --- | --- | --- | --- |
| **Spatial Hashing** | Grid 2D uniforme | Reduce vecinos O(n²) → O(k) | 10% memoria | Distribución uniforme |
| **Hierarchical Grids** | Multi-resolución | Adaptativo a densidad | 20% memoria | Distribución irregular |
| **Memory Pooling** | Pre-allocación | Evita malloc/free | 15% memoria | Creación/destrucción agentes |
| **Texture Memory** | Datos read-only | Cache automático | Ninguno | Mapas estáticos |
| **Constant Memory** | Parámetros globales | Broadcast eficiente | 64KB límite | Configuración simulación |

### 2.2 Paralelización del Pathfinding

#### 2.2.1 Algoritmo A\* Paralelo Jerárquico

\*\*Tabla 5: Implementación HPA\* (Hierarchical Path A\*)\*\*

| **Nivel** | **Granularidad** | **Algoritmo** | **Paralelización** | **Memoria** | **Latencia** |
| --- | --- | --- | --- | --- | --- |
| **Nivel 0** | Celdas (1m²) | A\* clásico | Por agente | 1MB/agente | <1ms |
| **Nivel 1** | Clusters (100m²) | A\* abstracto | Por cluster | 100KB/cluster | <10ms |
| **Nivel 2** | Sectores (1km²) | Dijkstra | Por sector | 10KB/sector | <100ms |
| **Nivel 3** | Regiones (10km²) | Floyd-Warshall | Secuencial | 1KB/región | <1s |

#### 2.2.2 Implementación Paralela CPU

class ParallelPathfinder {  
private:  
 ThreadPool threadPool;  
 HierarchicalGraph graph;  
 SpatialIndex spatialIndex;  
   
public:  
 std::vector<Path> findPaths(const std::vector<PathRequest>& requests) {  
 std::vector<std::future<Path>> futures;  
   
 // Agrupar requests por región para localidad  
 auto groupedRequests = groupByRegion(requests);  
   
 for (const auto& group : groupedRequests) {  
 // Lanzar trabajo asíncrono por grupo  
 auto future = threadPool.enqueue([this, group]() {  
 return processPathGroup(group);  
 });  
 futures.push\_back(std::move(future));  
 }  
   
 // Recolectar resultados  
 std::vector<Path> results;  
 for (auto& future : futures) {  
 auto groupResults = future.get();  
 results.insert(results.end(), groupResults.begin(), groupResults.end());  
 }  
   
 return results;  
 }  
   
private:  
 std::vector<Path> processPathGroup(const std::vector<PathRequest>& group) {  
 std::vector<Path> paths;  
   
 for (const auto& request : group) {  
 // Usar cache local para evitar contención  
 thread\_local PathCache cache;  
   
 Path path = findHierarchicalPath(request, cache);  
 paths.push\_back(path);  
 }  
   
 return paths;  
 }  
};

### 2.3 Paralelización del Sistema DES

#### 2.3.1 Arquitectura de Eventos Distribuidos

**Tabla 6: Estrategia de Paralelización DES**

| **Componente** | **Paralelización** | **Sincronización** | **Consistencia** | **Rendimiento** |
| --- | --- | --- | --- | --- |
| **Event Queue** | Particionado por tiempo | Barriers periódicos | Eventual | 100K eventos/sec |
| **Resource Management** | Sharding por tipo | Locks granulares | Fuerte | 50K ops/sec |
| **State Updates** | Copy-on-Write | MVCC | Snapshot | 200K reads/sec |
| **Statistics** | Reduce distribuido | MapReduce | Eventual | 1M agregaciones/sec |

#### 2.3.2 Implementación con Actor Model

class DESActor : public Actor {  
private:  
 EventQueue localQueue;  
 ResourceManager resourceMgr;  
 std::atomic<SimulationTime> localTime;  
   
public:  
 void processMessages() override {  
 while (true) {  
 Message msg = receiveMessage();  
   
 switch (msg.type) {  
 case EVENT\_SCHEDULE:  
 handleEventSchedule(msg);  
 break;  
   
 case RESOURCE\_REQUEST:  
 handleResourceRequest(msg);  
 break;  
   
 case SYNC\_BARRIER:  
 handleSyncBarrier(msg);  
 break;  
   
 case TERMINATION:  
 return;  
 }  
 }  
 }  
   
private:  
 void handleEventSchedule(const Message& msg) {  
 Event event = msg.getEvent();  
   
 if (event.timestamp >= localTime.load()) {  
 localQueue.schedule(event);  
 } else {  
 // Reenviar a actor responsable del tiempo  
 sendMessage(getTimeActor(event.timestamp), msg);  
 }  
 }  
   
 void handleResourceRequest(const Message& msg) {  
 ResourceRequest req = msg.getResourceRequest();  
   
 if (resourceMgr.canAllocate(req)) {  
 resourceMgr.allocate(req);  
 sendMessage(msg.sender, createResponse(SUCCESS));  
 } else {  
 sendMessage(msg.sender, createResponse(RESOURCE\_UNAVAILABLE));  
 }  
 }  
};

## 3. Arquitecturas de Hardware Específicas

### 3.1 Configuración de Cluster GPU

**Tabla 7: Especificaciones del Cluster GPU**

| **Componente** | **Especificación** | **Cantidad** | **Rendimiento** | **Costo (USD)** |
| --- | --- | --- | --- | --- |
| **GPU** | NVIDIA A100 80GB | 32 | 312 TFLOPS/GPU | $320,000 |
| **CPU** | AMD EPYC 7742 | 8 | 64 cores/CPU | $48,000 |
| **RAM** | DDR4-3200 ECC | 4TB | 3200 MT/s | $32,000 |
| **Storage** | NVMe SSD | 100TB | 7GB/s | $50,000 |
| **Network** | InfiniBand HDR | 8 ports | 200 Gb/s | $40,000 |
| **Power** | 80+ Platinum PSU | 16 | 95% efficiency | $8,000 |
| **Cooling** | Liquid cooling | 8 systems | 50kW capacity | $16,000 |
| **Total** | - | - | - | **$514,000** |

### 3.2 Configuración de Cluster CPU

**Tabla 8: Especificaciones del Cluster CPU**

| **Componente** | **Especificación** | **Cantidad** | **Rendimiento** | **Costo (USD)** |
| --- | --- | --- | --- | --- |
| **CPU** | Intel Xeon Platinum 8380 | 128 | 40 cores/CPU | $384,000 |
| **RAM** | DDR4-3200 ECC | 16TB | 3200 MT/s | $128,000 |
| **Storage** | NVMe SSD | 200TB | 7GB/s | $100,000 |
| **Network** | InfiniBand HDR | 32 ports | 200 Gb/s | $160,000 |
| **Switches** | InfiniBand Switch | 4 | 40 ports/switch | $80,000 |
| **Chassis** | 2U Servers | 64 | 2 CPUs/chassis | $128,000 |
| **Total** | - | - | - | **$980,000** |

### 3.3 Arquitectura de Red de Interconexión

**Tabla 9: Topología de Red HPC**

| **Capa** | **Tecnología** | **Ancho de Banda** | **Latencia** | **Topología** |
| --- | --- | --- | --- | --- |
| **Nivel 1** | InfiniBand HDR | 200 Gb/s | <1 μs | Fat-tree |
| **Nivel 2** | Ethernet 100G | 100 Gb/s | <10 μs | Spine-leaf |
| **Nivel 3** | Ethernet 10G | 10 Gb/s | <100 μs | Mesh |
| **Storage** | Fibre Channel | 32 Gb/s | <5 μs | SAN fabric |

## 4. Benchmarks y Métricas de Rendimiento

### 4.1 Benchmarks de Escalabilidad

**Tabla 10: Resultados de Benchmarks de Escalabilidad**

| **Configuración** | **Agentes** | **Tiempo Real** | **Tiempo Simulado** | **Speedup** | **Eficiencia** |
| --- | --- | --- | --- | --- | --- |
| **CPU 1 core** | 1,000 | 60 min | 30 min | 1.0x | 100% |
| **CPU 8 cores** | 8,000 | 65 min | 30 min | 7.4x | 92% |
| **CPU 64 cores** | 64,000 | 75 min | 30 min | 51.2x | 80% |
| **GPU 1 device** | 100,000 | 45 min | 30 min | 85.3x | 85% |
| **GPU 8 devices** | 800,000 | 55 min | 30 min | 545.5x | 68% |
| **GPU 32 devices** | 3,200,000 | 90 min | 30 min | 1,280.0x | 40% |

### 4.2 Análisis de Rendimiento por Componente

**Tabla 11: Profiling Detallado de Componentes**

| **Componente** | **% Tiempo CPU** | **% Tiempo GPU** | **Memoria (GB)** | **I/O (MB/s)** |
| --- | --- | --- | --- | --- |
| **Social Force Model** | 15% | 75% | 24 | 100 |
| **Pathfinding** | 45% | 5% | 8 | 50 |
| **DES Events** | 25% | 0% | 4 | 200 |
| **Collision Detection** | 5% | 15% | 2 | 10 |
| **Visualization** | 2% | 5% | 6 | 500 |
| **I/O & Sync** | 8% | 0% | 1 | 1000 |

### 4.3 Métricas de Eficiencia Energética

**Tabla 12: Consumo Energético por Configuración**

| **Configuración** | **Potencia (kW)** | **Agentes/Watt** | **FLOPS/Watt** | **Costo/hora (USD)** |
| --- | --- | --- | --- | --- |
| **CPU Only** | 25 | 2,560 | 400 GFLOPS | $3.50 |
| **GPU Only** | 35 | 22,857 | 2,800 GFLOPS | $4.90 |
| **Híbrido** | 45 | 71,111 | 6,900 GFLOPS | $6.30 |
| **Cloud (AWS)** | Variable | 15,000 | 1,200 GFLOPS | $12.00 |

## 5. Optimizaciones Algorítmicas Avanzadas

### 5.1 Técnicas de Reducción de Complejidad

**Tabla 13: Técnicas de Optimización Algorítmica**

| **Técnica** | **Reducción Complejidad** | **Precisión** | **Implementación** | **Casos de Uso** |
| --- | --- | --- | --- | --- |
| **Spatial Hashing** | O(n²) → O(n) | 100% | Grid uniforme | Distribución uniforme |
| **Hierarchical Methods** | O(n²) → O(n log n) | 95% | Octree/Quadtree | Distribución irregular |
| **Fast Multipole** | O(n²) → O(n) | 99% | Expansión multipolar | Fuerzas largo alcance |
| **Approximate NN** | O(n log n) → O(n) | 90% | LSH/Random projection | Búsqueda vecinos |
| **Level of Detail** | Variable | 80-100% | Adaptativo | Visualización |

### 5.2 Algoritmo Fast Multipole Method (FMM)

class FastMultipoleMethod {  
private:  
 struct MultipoleExpansion {  
 std::complex<double> coefficients[MAX\_TERMS];  
 Point2D center;  
 double radius;  
 };  
   
 struct LocalExpansion {  
 std::complex<double> coefficients[MAX\_TERMS];  
 Point2D center;  
 double radius;  
 };  
   
 std::vector<MultipoleExpansion> multipoleExpansions;  
 std::vector<LocalExpansion> localExpansions;  
 QuadTree spatialTree;  
   
public:  
 void calculateForces(std::vector<Agent>& agents) {  
 // Paso 1: Construir árbol espacial  
 spatialTree.build(agents);  
   
 // Paso 2: Formar expansiones multipolares (bottom-up)  
 formMultipoleExpansions(spatialTree.getRoot());  
   
 // Paso 3: Convertir M2L (multipole-to-local)  
 convertM2L(spatialTree.getRoot());  
   
 // Paso 4: Propagar expansiones locales (top-down)  
 propagateLocalExpansions(spatialTree.getRoot());  
   
 // Paso 5: Evaluar fuerzas finales  
 evaluateLocalForces(agents);  
 }  
   
private:  
 void formMultipoleExpansions(QuadTreeNode\* node) {  
 if (node->isLeaf()) {  
 // Formar expansión multipolar directamente de las partículas  
 MultipoleExpansion& expansion = multipoleExpansions[node->id];  
 expansion.center = node->center;  
 expansion.radius = node->radius;  
   
 for (int k = 0; k < MAX\_TERMS; k++) {  
 expansion.coefficients[k] = 0.0;  
   
 for (const Agent& agent : node->agents) {  
 std::complex<double> z = std::complex<double>(  
 agent.position.x - expansion.center.x,  
 agent.position.y - expansion.center.y  
 );  
   
 expansion.coefficients[k] += agent.mass \* std::pow(z, k);  
 }  
 }  
 } else {  
 // Formar expansión multipolar de hijos  
 for (QuadTreeNode\* child : node->children) {  
 formMultipoleExpansions(child);  
 }  
   
 // Combinar expansiones de hijos  
 combineMultipoleExpansions(node);  
 }  
 }  
};

### 5.3 Optimización de Acceso a Memoria

**Tabla 14: Patrones de Acceso a Memoria**

| **Patrón** | **Cache Hit Rate** | **Bandwidth Utilization** | **Latencia** | **Optimización** |
| --- | --- | --- | --- | --- |
| **Sequential** | 95% | 90% | 100 cycles | Prefetching |
| **Strided** | 70% | 60% | 200 cycles | Blocking |
| **Random** | 30% | 20% | 400 cycles | Spatial locality |
| **Indirect** | 25% | 15% | 500 cycles | Gather/scatter |

## 6. Tolerancia a Fallos y Recuperación

### 6.1 Estrategias de Checkpoint

**Tabla 15: Estrategias de Checkpoint y Recuperación**

| **Estrategia** | **Frecuencia** | **Overhead** | **Tiempo Recuperación** | **Almacenamiento** |
| --- | --- | --- | --- | --- |
| **Full Checkpoint** | Cada 1 hora | 15% | 10 minutos | 100% estado |
| **Incremental** | Cada 10 min | 5% | 5 minutos | 10% estado |
| **Hierarchical** | Adaptativo | 8% | 7 minutos | 30% estado |
| **In-Memory** | Cada 1 min | 2% | 1 minuto | RAM only |

### 6.2 Implementación de Checkpoint Distribuido

class DistributedCheckpoint {  
private:  
 struct CheckpointData {  
 std::vector<Agent> agents;  
 std::vector<Event> events;  
 SimulationTime currentTime;  
 std::map<std::string, double> statistics;  
 uint64\_t checksum;  
 };  
   
 MPI\_Comm communicator;  
 int rank, size;  
 std::string checkpointDir;  
   
public:  
 void createCheckpoint(const SimulationState& state) {  
 CheckpointData data;  
 data.agents = state.getAgents();  
 data.events = state.getEvents();  
 data.currentTime = state.getCurrentTime();  
 data.statistics = state.getStatistics();  
 data.checksum = calculateChecksum(data);  
   
 // Checkpoint coordinado  
 MPI\_Barrier(communicator);  
   
 // Escribir datos locales  
 std::string filename = checkpointDir + "/checkpoint\_" +   
 std::to\_string(rank) + ".bin";  
 writeCheckpointFile(filename, data);  
   
 // Verificar integridad  
 MPI\_Barrier(communicator);  
 if (rank == 0) {  
 verifyCheckpointIntegrity();  
 }  
 }  
   
 SimulationState restoreCheckpoint() {  
 std::string filename = checkpointDir + "/checkpoint\_" +   
 std::to\_string(rank) + ".bin";  
   
 CheckpointData data = readCheckpointFile(filename);  
   
 // Verificar checksum  
 uint64\_t expectedChecksum = calculateChecksum(data);  
 if (data.checksum != expectedChecksum) {  
 throw CheckpointCorruptedException();  
 }  
   
 // Restaurar estado  
 SimulationState state;  
 state.setAgents(data.agents);  
 state.setEvents(data.events);  
 state.setCurrentTime(data.currentTime);  
 state.setStatistics(data.statistics);  
   
 return state;  
 }  
};

## 7. Casos de Uso y Validación

### 7.1 Casos de Uso Específicos

**Tabla 16: Casos de Uso de Validación HPC**

| **Caso de Uso** | **Población** | **Duración** | **Recursos HPC** | **Tiempo Objetivo** | **Métricas Clave** |
| --- | --- | --- | --- | --- | --- |
| **Tsunami Valparaíso** | 300,000 | 2 horas | 16 GPUs | 30 minutos | Tiempo evacuación, congestión |
| **Sismo Santiago** | 1,000,000 | 4 horas | 32 GPUs | 1 hora | Rutas óptimas, recursos |
| **Erupción Volcánica** | 150,000 | 6 horas | 8 GPUs | 45 minutos | Dispersión ceniza, salud |
| **Incendio Forestal** | 50,000 | 8 horas | 4 GPUs | 20 minutos | Propagación fuego, humo |
| **Simulacro Nacional** | 2,000,000 | 3 horas | 64 GPUs | 2 horas | Coordinación, comunicación |

### 7.2 Métricas de Validación

**Tabla 17: KPIs de Rendimiento HPC**

| **Categoría** | **Métrica** | **Valor Objetivo** | **Método Medición** | **Frecuencia** |
| --- | --- | --- | --- | --- |
| **Rendimiento** | Agentes/segundo | >100,000 | Profiler interno | Continua |
| **Escalabilidad** | Eficiencia paralela | >70% | Benchmarks | Semanal |
| **Memoria** | Utilización RAM | <80% | Monitoreo sistema | Continua |
| **Red** | Throughput | >80% capacidad | Análisis tráfico | Continua |
| **Energía** | PUE (Power Usage) | <1.5 | Medidores hardware | Diaria |
| **Disponibilidad** | Uptime | >99.5% | Monitoreo servicios | Continua |

### 7.3 Validación Experimental

**Tabla 18: Resultados Experimentales**

| **Experimento** | **Configuración** | **Resultado** | **Desviación** | **Validación** |
| --- | --- | --- | --- | --- |
| **Weak Scaling** | 1K-1M agentes | Eficiencia 75% | ±5% | Ley de Gustafson |
| **Strong Scaling** | 100K agentes fijos | Speedup 45x | ±10% | Ley de Amdahl |
| **Memory Scaling** | 1GB-100GB datos | Lineal | ±3% | Modelo teórico |
| **Network Scaling** | 1-64 nodos | Overhead 15% | ±8% | Modelo bisección |

## 8. Integración con Infraestructura Cloud

### 8.1 Arquitectura Híbrida On-Premise/Cloud

**Tabla 19: Estrategia de Cloud Bursting**

| **Escenario** | **Recursos Local** | **Cloud Bursting** | **Latencia** | **Costo** |
| --- | --- | --- | --- | --- |
| **Operación Normal** | 100% | 0% | <1ms | $0/hora |
| **Pico Demanda** | 100% | 200% | <50ms | $50/hora |
| **Emergencia** | 100% | 500% | <100ms | $200/hora |
| **Investigación** | 50% | 300% | <200ms | $80/hora |

### 8.2 Implementación Kubernetes

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: pigem-simulation  
spec:  
 replicas: 8  
 selector:  
 matchLabels:  
 app: pigem-simulation  
 template:  
 metadata:  
 labels:  
 app: pigem-simulation  
 spec:  
 nodeSelector:  
 accelerator: nvidia-tesla-v100  
 containers:  
 - name: simulation-engine  
 image: pigem/simulation:latest  
 resources:  
 requests:  
 memory: "32Gi"  
 cpu: "8"  
 nvidia.com/gpu: "1"  
 limits:  
 memory: "64Gi"  
 cpu: "16"  
 nvidia.com/gpu: "2"  
 env:  
 - name: CUDA\_VISIBLE\_DEVICES  
 value: "0,1"  
 - name: MPI\_RANKS  
 value: "8"  
 volumeMounts:  
 - name: shared-storage  
 mountPath: /data  
 volumes:  
 - name: shared-storage  
 persistentVolumeClaim:  
 claimName: pigem-pvc  
---  
apiVersion: v1  
kind: Service  
metadata:  
 name: pigem-simulation-service  
spec:  
 selector:  
 app: pigem-simulation  
 ports:  
 - port: 8080  
 targetPort: 8080  
 type: LoadBalancer

## 9. Consideraciones de Costos

### 9.1 Análisis de Costo Total de Propiedad (TCO)

**Tabla 20: TCO a 5 años (USD)**

| **Componente** | **CAPEX** | **OPEX/año** | **TCO 5 años** | **% del Total** |
| --- | --- | --- | --- | --- |
| **Hardware** | $1,500,000 | $150,000 | $2,250,000 | 60% |
| **Software** | $200,000 | $100,000 | $700,000 | 19% |
| **Personal** | $0 | $120,000 | $600,000 | 16% |
| **Infraestructura** | $100,000 | $30,000 | $250,000 | 7% |
| **Mantenimiento** | $0 | $50,000 | $250,000 | 7% |
| **Total** | **$1,800,000** | **$450,000** | **$4,050,000** | **100%** |

### 9.2 Comparación Cloud vs On-Premise

**Tabla 21: Análisis Costo-Beneficio**

| **Modelo** | **Costo Inicial** | **Costo/hora** | **Flexibilidad** | **Control** | **Recomendación** |
| --- | --- | --- | --- | --- | --- |
| **On-Premise** | Alto | Bajo | Baja | Alto | Uso intensivo |
| **Cloud Público** | Bajo | Alto | Alta | Bajo | Uso esporádico |
| **Cloud Privado** | Medio | Medio | Media | Medio | Uso mixto |
| **Híbrido** | Medio | Variable | Muy Alta | Alto | **Recomendado** |

## 10. Bibliografía Especializada

### 10.1 Referencias en Computación Paralela

* Pacheco, P. (2021). *An Introduction to Parallel Programming* (2nd ed.). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-374260-5.00001-4
* Rauber, T., & Rünger, G. (2023). *Parallel Programming: for Multicore and Cluster Systems* (3rd ed.). Springer. https://doi.org/10.1007/978-3-662-65471-1
* Grama, A., Karypis, G., Kumar, V., & Gupta, A. (2003). *Introduction to Parallel Computing* (2nd ed.). Addison-Wesley.

### 10.2 Referencias en GPU Computing

* Kirk, D., & Hwu, W. (2022). *Programming Massively Parallel Processors: A Hands-on Approach* (4th ed.). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-811986-0.00001-4
* Sanders, J., & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional.
* Cheng, J., Grossman, M., & McKercher, T. (2014). *Professional CUDA C Programming*. Wrox Press.

### 10.3 Referencias en HPC para Simulación

* Dongarra, J., Foster, I., Fox, G., Gropp, W., Kennedy, K., Torczon, L., & White, A. (2003). *Sourcebook of Parallel Computing*. Morgan Kaufmann.
* Karniadakis, G., & Kirby, R. M. (2003). *Parallel Scientific Computing in C++ and MPI*. Cambridge University Press.
* Mattson, T., Sanders, B., & Massingill, B. (2004). *Patterns for Parallel Programming*. Addison-Wesley Professional.

### 10.4 Referencias en Optimización de Rendimiento

* Hennessy, J. L., & Patterson, D. A. (2019). *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-811905-1.00001-9
* Drepper, U. (2007). *What Every Programmer Should Know About Memory*. Red Hat Inc.
* Agner, F. (2022). *Optimizing Software in C++: An Optimization Guide for Windows, Linux and Mac Platforms*. Copenhagen University College of Engineering.

**Pablo Antonio Jordán González**  
Estudiante Doctorado Ciencias de la Ingeniería Mención Informática  
Universidad de Santiago de Chile  
Director de I+D HealthPixel Spa.