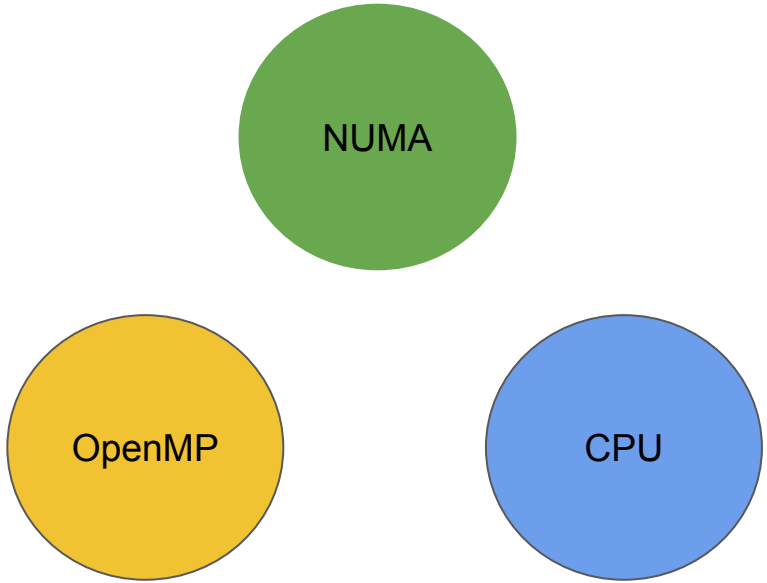
Proyecto 6: Afinidad NUMA Dinámica y Políticas OpenMP.

Juan Sebastián Otero Vega
David Fernando Naranjo Rangel

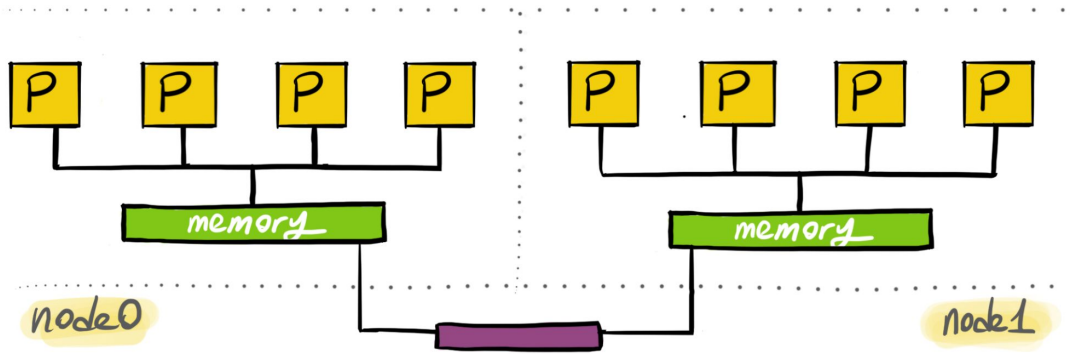
Universidad Industrial de Santander

Título:

Implementar un runtime basado en métricas de acceso a memoria para el ajuste políticas de afinidad de hilos y asignación de memoria NUMA en aplicaciones OpenMp.



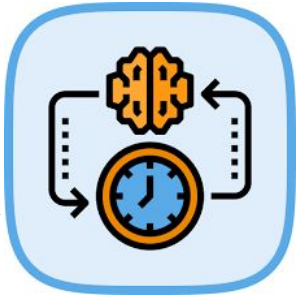
Que es NUMA?



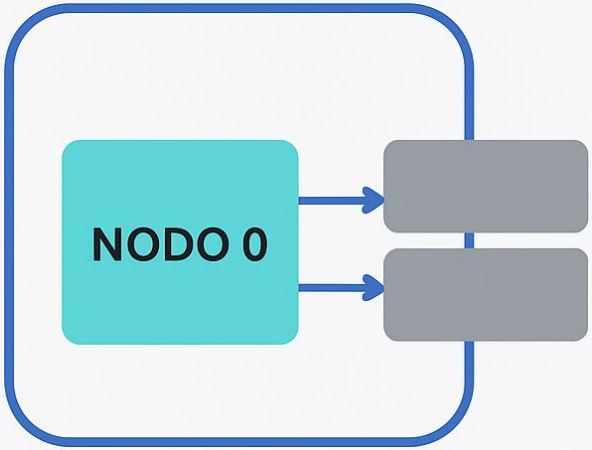
numactl

numastat

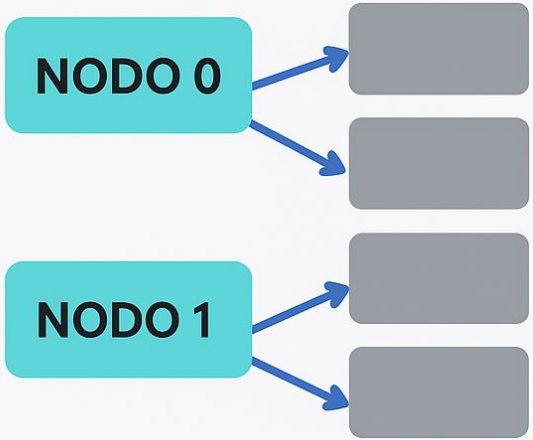
libnuma



POLÍTICAS NUMA



membind



interleave

Tabla de uso de memoria

```
[dfnaranjor@ExaDELL ~]$ numastat -p 2818009

Per-node process memory usage (in MBs) for PID 2818009 (BFSgraph128)
                           Node 0          Node 1           Total
                  --------------- --------------- ---------------
Huge                         0.00            0.00            0.00
Heap                     91718.86         7478.71        99197.57
Stack                        0.03            0.00            0.03
Private                  13604.63          139.36        13743.99
----------------  --------------- --------------- ---------------
Total                   105323.52         7618.07       112941.59
```

Sin uso de políticas NUMA

Uso de políticas NUMA

```
[dfnaranjor@ExaDELL ~]$ numastat -p 2819370

Per-node process memory usage (in MBs) for PID 2819370 (BFSgraph64)
                           Node 0          Node 1           Total
                  --------------- --------------- ---------------
Huge                         0.00            0.00            0.00
Heap                         0.00        99212.50        99212.50
Stack                        0.00            0.03            0.03
Private                      2.26        13397.51        13399.77
----------------  --------------- --------------- ---------------
Total                        2.26       112610.04       112612.30
```

OpenMP

OMP_PLACES

Encargada de definir los lugares donde se ejecutarán los hilos OpenMP:

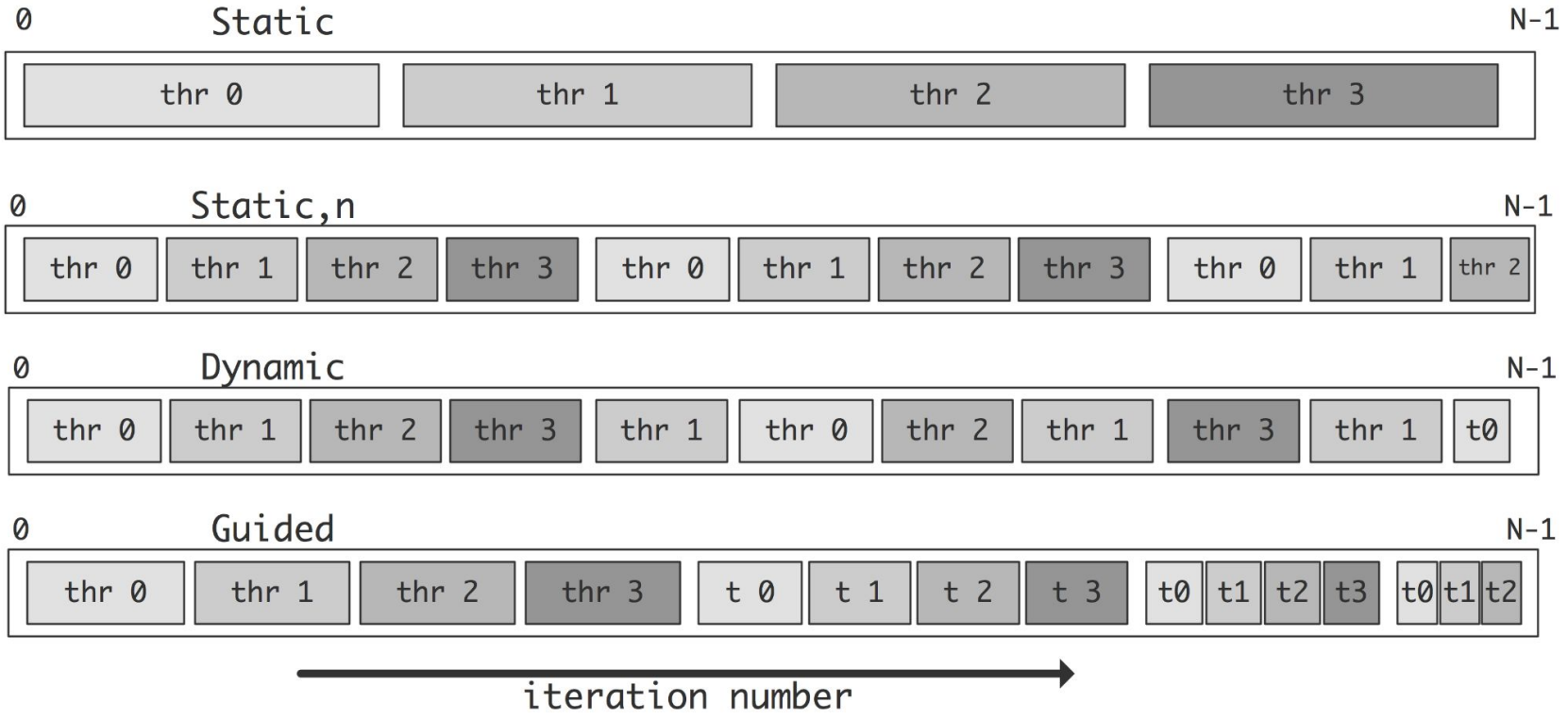
- Threads
- Cores
- Sockets

OMP_PROC_BIND

Encargada de manejar la forma en que se vinculan los hilos OpenMP a sus lugares:

- Close
- Spread
- Master

Un poco de load balance…



```cpp
#include <omp.h>
#include <vector>
#include <atomic>
#include <random>
#include <iostream>
#include <algorithm>


const int N = 10000000; // 10M nodos
const int AVG_DEGREE = 50;

int main() {
    // ================================================================ FASE 1: CONSTRUCCIÓN DEL GRAFO
    // ================================================================

    // Reservar memoria para el grafo (vector de vectores)
    std::vector<std::vector<int>> graph(N);

    // Generador de números aleatorios con semilla fija (reproducibilidad)
    std::mt19937 gen(42);
    std::uniform_int_distribution<> dis(0, N-1);

    // Construir grafo aleatorio: cada nodo tiene AVG_DEGREE vecinos
    std::cout << "Construyendo grafo..." << std::endl;
        for(int i = 0; i < N; i++) {
                graph[i].reserve(AVG_DEGREE); // Pre-reservar espacio (eficiencia)
                for(int j = 0; j < AVG_DEGREE; j++) {
                        graph[i].push_back(dis(gen)); // Agregar vecino aleatorio
                }
        }
```

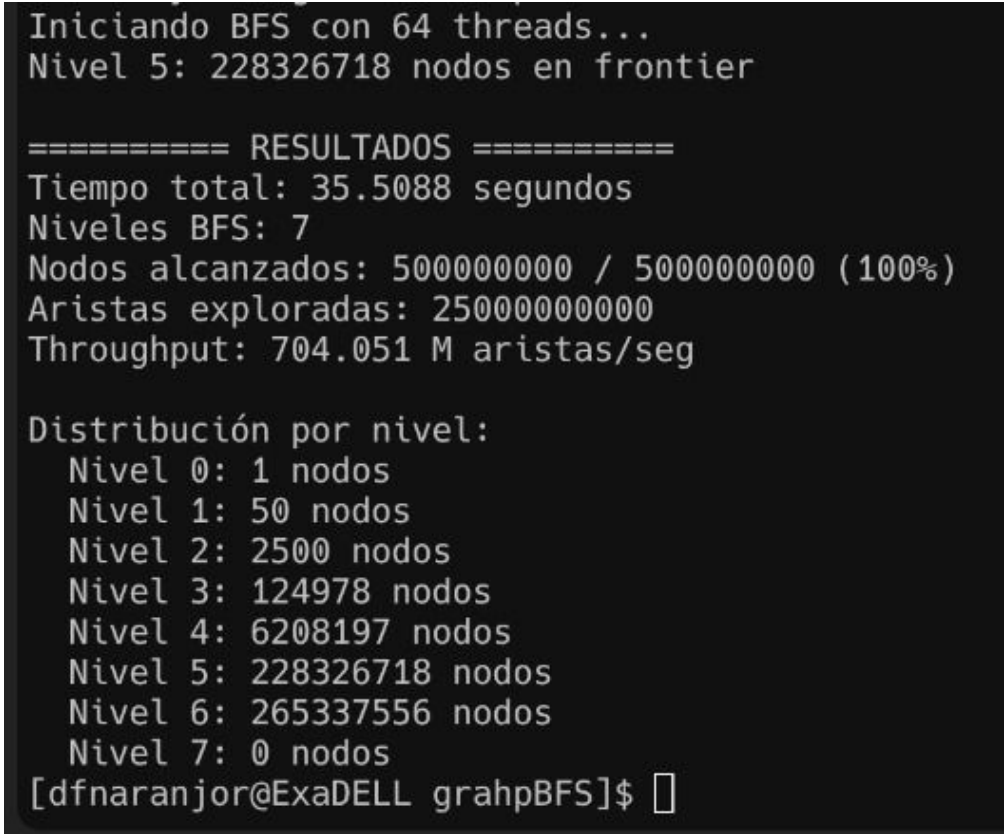
```cpp
// ============================================================ FASE 2: INICIALIZACIÓN DE ESTRUCTURAS BFS
// ============================================================

// Vector de distancias: -1 = no visitado, >=0 = nivel BFS
std::vector<int> distance(N, -1);
distance[0] = 0; // El nodo 0 es la raíz (distancia 0)

// Vector atómico para marcar visitados sin race conditions std::atomic<bool> no funciona en vector, usamos vector de atomics manual
std::vector<std::atomic<bool>> visited(N);
    for(int i = 0; i < N; i++) {
            visited[i].store(false, std::memory_order_relaxed); // Inicializar a false
    }
visited[0].store(true, std::memory_order_relaxed); // Marcar raíz como visitada

// Frontier actual (nodos en el nivel actual del BFS)
std::vector<int> frontier = {0};
```

```
// ============================================================ FASE 3: CONFIGURACIÓN OPENMP
// ============================================================

omp_set_num_threads(64); // Establecer número de threads

// Variables para estadísticas
int level = 0;
long long total_edges_explored = 0;

std::cout << "Iniciando BFS con " << omp_get_max_threads() << " threads..." << std::endl; double start = omp_get_wtime(); // Tiempo inicial
```

```
// ============================================================ FASE 4: BFS PARALELO (BUCLE PRINCIPAL)
// ============================================================

while(!frontier.empty()) { // Mientras haya nodos por explorar

    // Vector para el siguiente nivel (next_frontier) Tamaño estimado: peor caso es frontier_size * AVG_DEGREE
    std::vector<int> next_frontier; next_frontier.reserve(frontier.size() * AVG_DEGREE / 10);

    // Contador atómico para saber dónde insertar en next_frontier
    std::atomic<size_t> next_frontier_size(0);

    // Buffer temporal grande para evitar sincronización Cada thread escribe aquí sin locks
    std::vector<int> temp_buffer(frontier.size() * AVG_DEGREE);


    // ------------------------------------------------------- REGIÓN PARALELA -------------------------------------------------------
    #pragma omp parallel
    {
        // Cada thread tiene su propio vector local (sin sincronización)
        std::vector<int> local_frontier;
        local_frontier.reserve(1000); // Capacidad inicial

        // ------------------------------------------------------- PARALELIZAR EXPLORACIÓN DE FRONTIER schedule(dynamic, 64): distribución
        // dinámica en chunks de 64 Bueno para balance de carga cuando nodos tienen grados variables
        // -----------------------------------------------------------
        #pragma omp for schedule(dynamic, 64) reduction(+:total_edges_explored)
        for(size_t i = 0; i < frontier.size(); i++) { int node = frontier[i]; // Nodo actual a explorar

            // Explorar todos los vecinos del nodo
            for(int neighbor : graph[node]) { total_edges_explored++; // Contar arista explorada

                // ------------------------------------------------- INTENTO ATÓMICO DE MARCAR COMO VISITADO compare_exchange_strong: compara y
                // cambia atómicamente Si visited[neighbor] es false, lo cambia a true y retorna true Si ya era true, retorna false (otro thread
                // lo visitó primero) -------------------------------------------------
                bool expected = false;
                if(visited[neighbor].compare_exchange_strong( expected, // Valor esperado (false)
                    true, // Nuevo valor (true)
                    std::memory_order_release, // Memoria order para escritura
                    std::memory_order_relaxed)) { // Memoria order si falla

                    // Solo este thread logró marcar el nodo
                    distance[neighbor] = level + 1; // Asignar distancia
                    local_frontier.push_back(neighbor); // Agregar a frontier local
                }
                // Si compare_exchange retornó false, otro thread ya procesó este nodo
            }
        }
```

```
        }

        // ------------------------------------------------------- FUSIÓN DE FRONTIERS LOCALES EN GLOBAL Sección crítica solo para fusionar
        // vectores locales (mucho más eficiente que critical por cada nodo) ------------------------------------------------------
        #pragma omp critical
        { next_frontier.insert(next_frontier.end(),
            local_frontier.begin(),
            local_frontier.end());
        }
    }
    // FIN REGIÓN PARALELA


    // ------------------------------------------------------- ACTUALIZAR PARA SIGUIENTE ITERACIÓN
    // ------------------------------------------------------
    frontier = std::move(next_frontier); // move: evita copiar, transfiere ownership
    level++; // Incrementar nivel BFS

    // Imprimir progreso cada 5 niveles
    if(level % 5 == 0) {
            std::cout << "Nivel " << level << ": " << frontier.size() << " nodos en frontier" << std::endl;
    }
}
```

```
    // ============================================================ FASE 5: RESULTADOS Y ESTADÍSTICAS
    // ============================================================

    double end = omp_get_wtime();
    double elapsed = end - start;

    // Contar nodos alcanzados
    long long nodes_reached = 0;
    #pragma omp parallel for reduction(+:nodes_reached)
    for(int i = 0; i < N; i++) {
        if(distance[i] != -1) nodes_reached++;
    }

    std::cout << "\n========== RESULTADOS ==========" << std::endl; std::cout << "Tiempo total: " << elapsed << " segundos" << std::endl; std::cout <<
    "Niveles BFS: " << level << std::endl; std::cout << "Nodos alcanzados: " << nodes_reached << " / " << N
              << " (" << (100.0*nodes_reached/N) << "%)" << std::endl; std::cout << "Aristas exploradas: " << total_edges_explored << std::endl;
    std::cout << "Throughput: " << (total_edges_explored/elapsed/1e6)
              << " M aristas/seg" << std::endl;

    // Calcular distribución de distancias
    std::vector<int> distance_histogram(level + 1, 0);
        for(int i = 0; i < N; i++) {
                if(distance[i] != -1) {
                        distance_histogram[distance[i]]++;
        }
    }

    std::cout << "\nDistribución por nivel:" << std::endl;
        for(int i = 0; i <= std::min(10, level); i++) {
                std::cout << " Nivel " << i << ": " <<
                distance_histogram[i] << " nodos" << std::endl;
    }

    return 0;
}
```



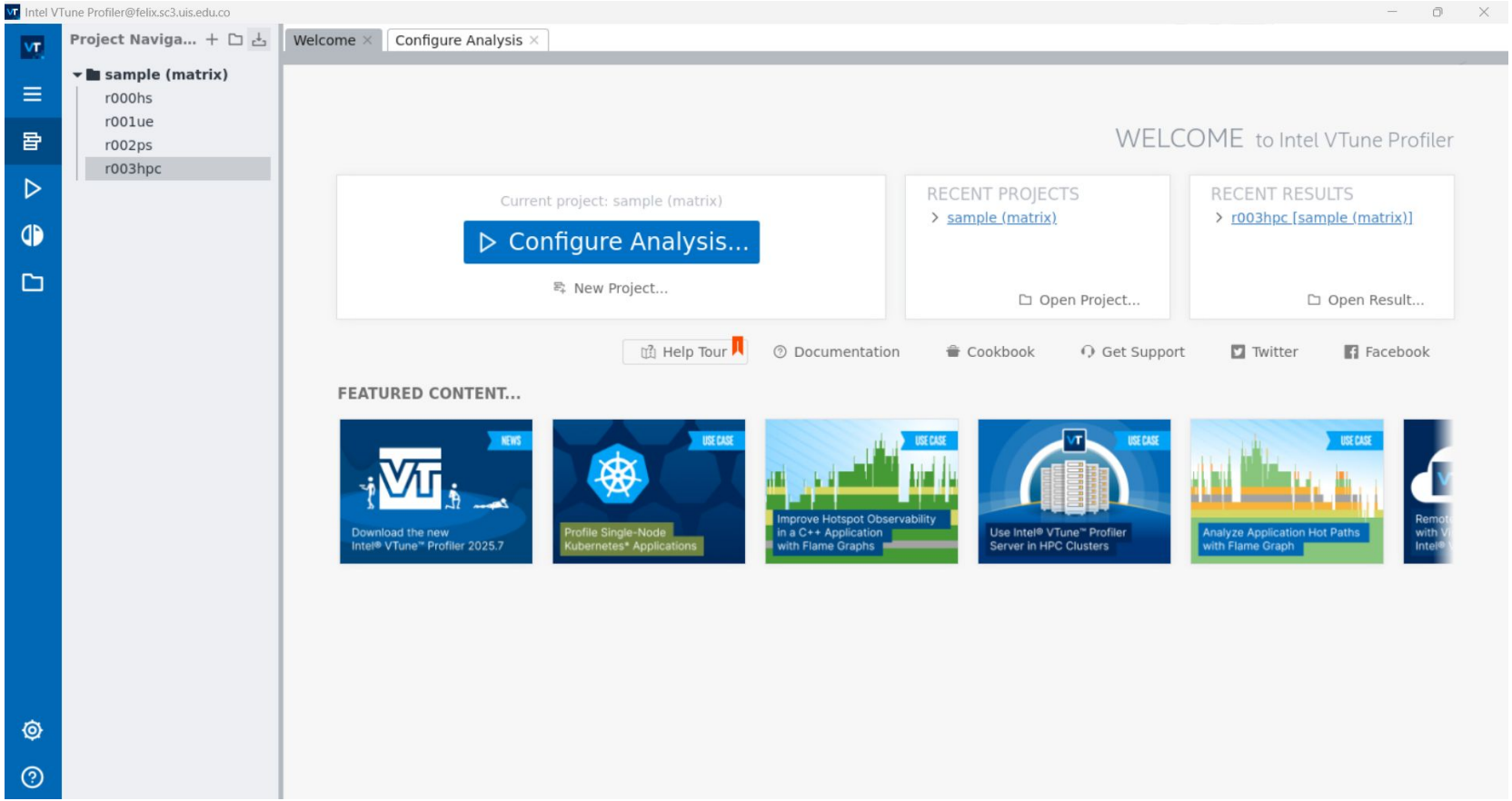
64 hilos de un nodo NUMA

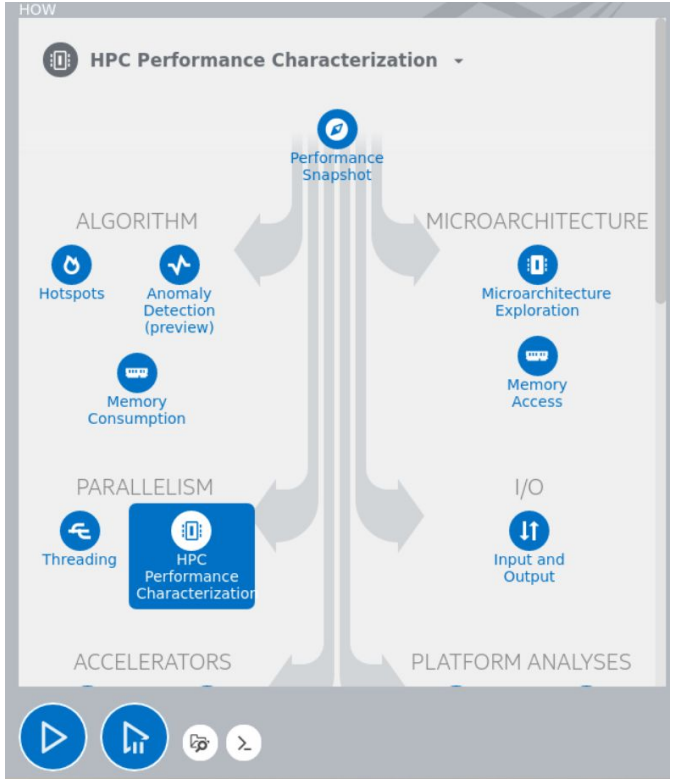
Usando políticas NUMA

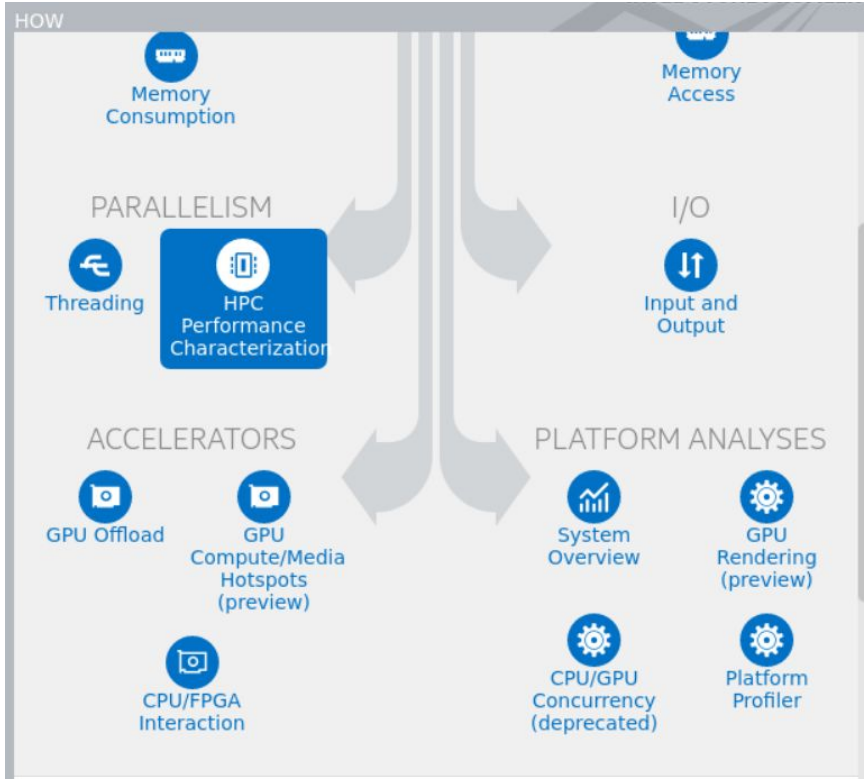
```
[dfnaranjor@ExaDELL grahpBFS]$ OMP_PLACES=threads OMP_PROC_BIND=close numactl --cpun
odebind=1 --membind=1 ./BFSgraph128
Construyendo grafo...
Iniciando BFS con 128 threads...
Nivel 5: 228326718 nodos en frontier

========== RESULTADOS ==========
Tiempo total: 35.8932 segundos
Niveles BFS: 7
Nodos alcanzados: 500000000 / 500000000 (100%)
Aristas exploradas: 25000000000
Throughput: 696.512 M aristas/seg

Distribución por nivel:
  Nivel 0: 1 nodos
  Nivel 1: 50 nodos
  Nivel 2: 2500 nodos
  Nivel 3: 124978 nodos
  Nivel 4: 6208197 nodos
  Nivel 5: 228326718 nodos
  Nivel 6: 265337556 nodos
  Nivel 7: 0 nodos
```

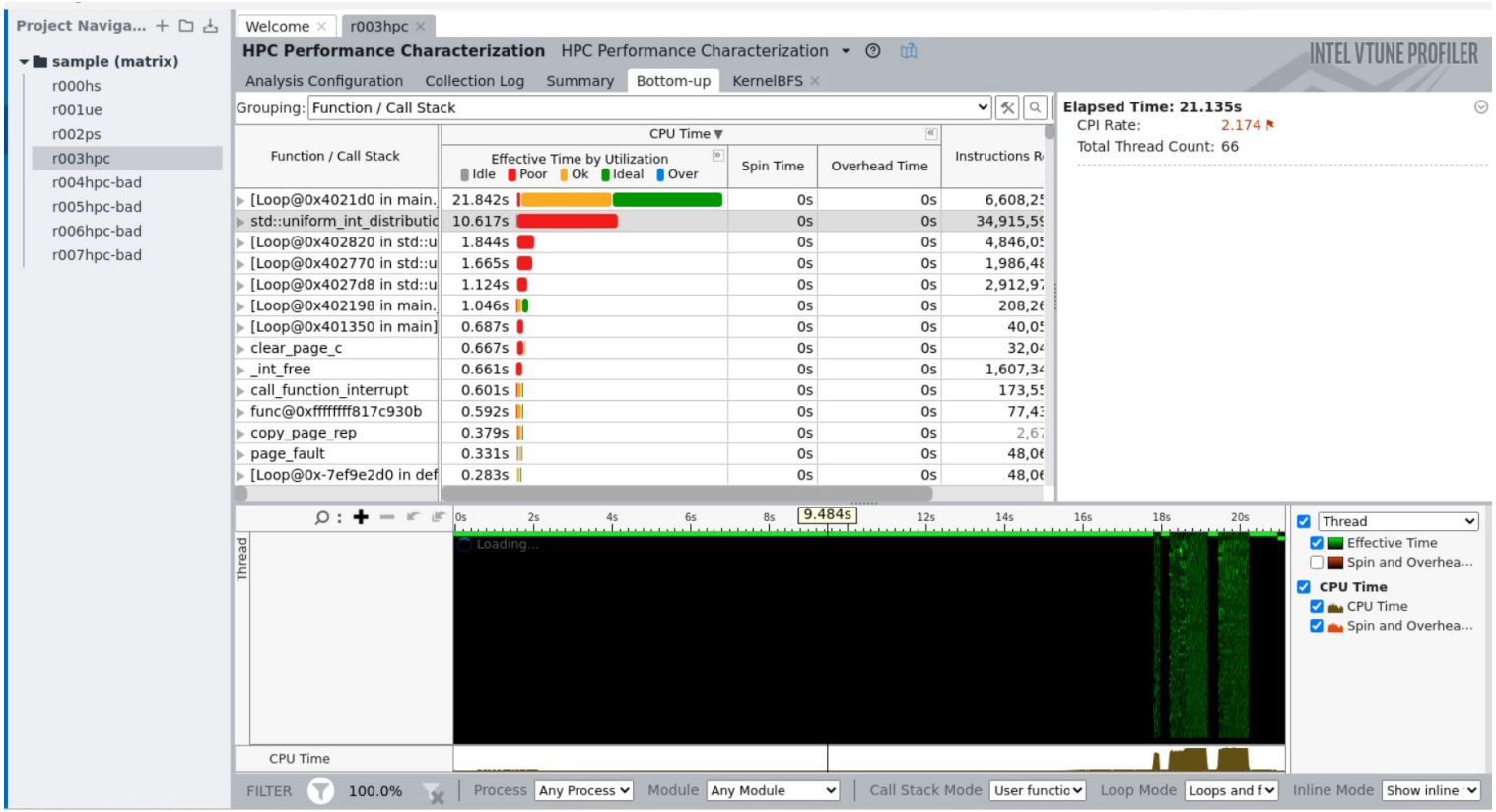
128 hilos de un nodo NUMA

Intel Vtune

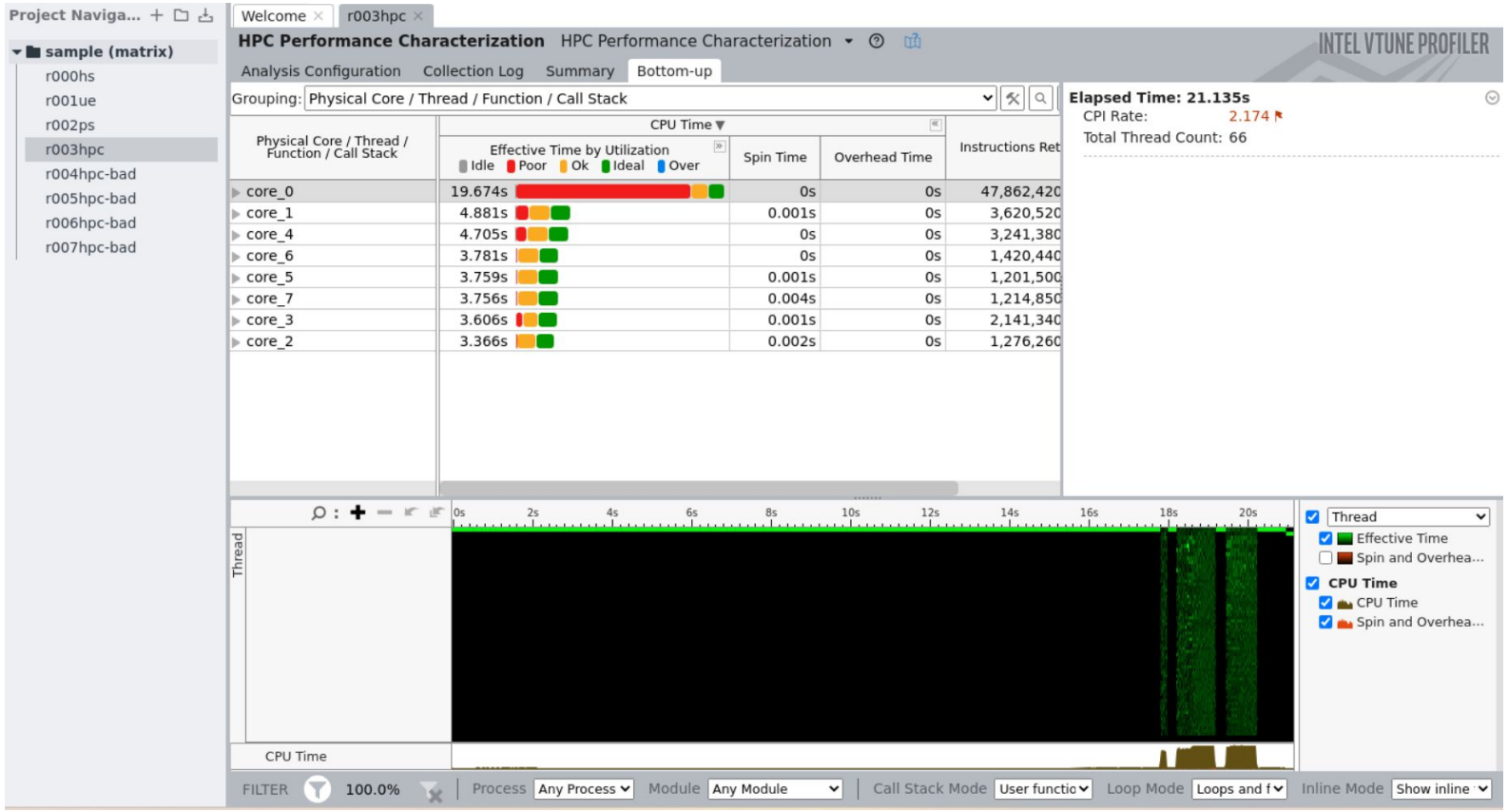

Intel VTune Profiler@felix.sc3.uis.edu.co
Project Naviga...
sample (matrix)
r000hs
r001ue
r002ps
r003hpc
Welcome
Configure Analysis
WELCOME to Intel VTune Profiler
Current project: sample (matrix)
Configure Analysis...
New Project...
RECENT PROJECTS
sample (matrix)
Open Project...
RECENT RESULTS
r003hpc [sample (matrix)]
Open Result...
Help Tour
Documentation
Cookbook
Get Support
Twitter
Facebook
FEATURED CONTENT...
NEWS
Download the new Intel® VTune™ Profiler 2025.7
USE CASE
Profile Single-Node Kubernetes* Applications
USE CASE
Improve Hotspot Observability in a C++ Application with Flame Graphs
USE CASE
Use Intel® VTune™ Profiler Server in HPC Clusters
USE CASE
Analyze Application Hot Paths with Flame Graph

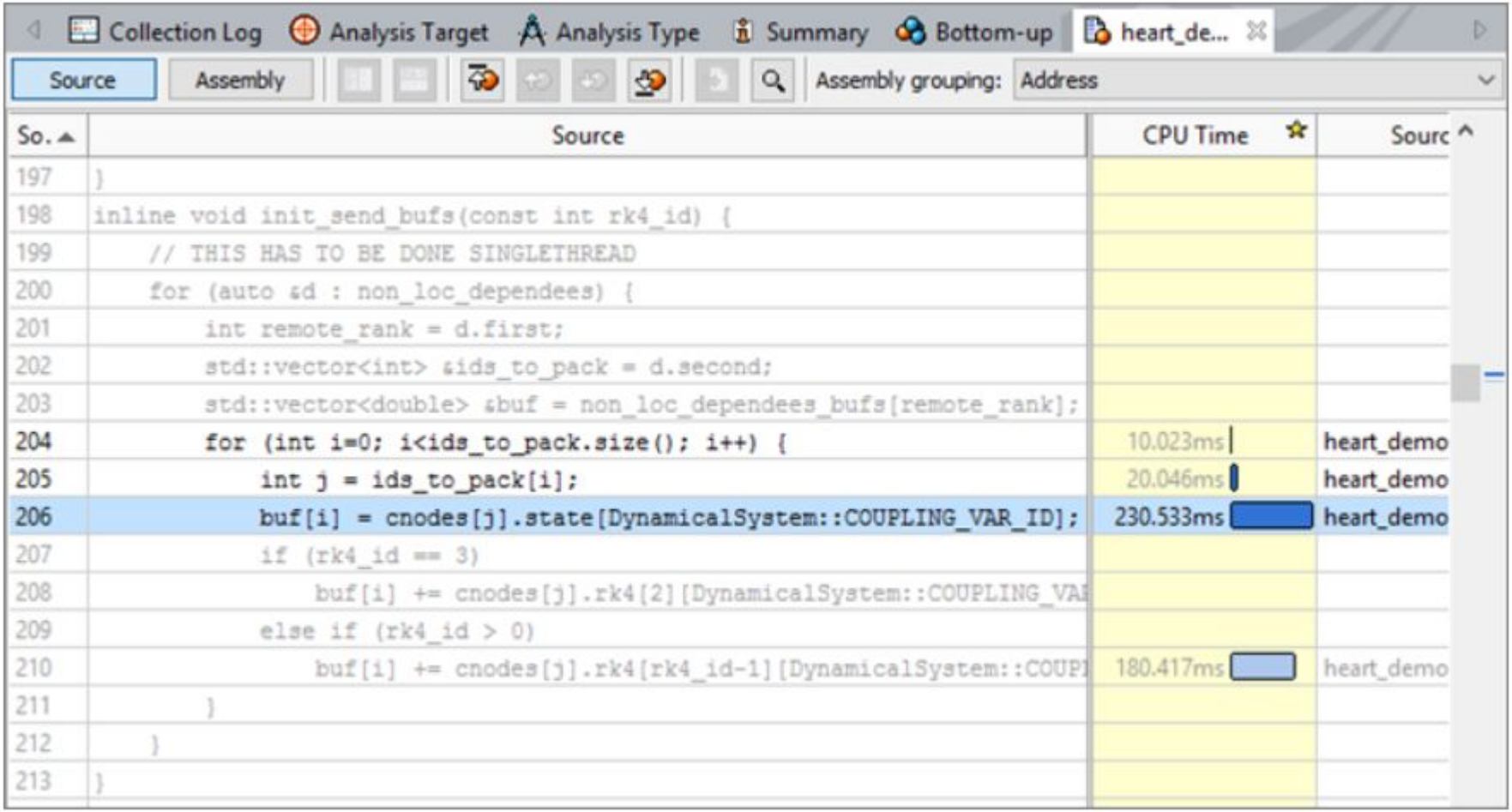

HOW
HPC Performance Characterization
Performance Snapshot
ALGORITHM
Hotspots
Anomaly Detection (preview)
Memory Consumption
MICROARCHITECTURE
Microarchitecture Exploration
Memory Access
PARALLELISM
Threading
HPC Performance Characterizatior
I/O
Input and Output
ACCELERATORS
PLATFORM ANALYSES


HOW
Memory Consumption
Memory Access
PARALLELISM
Threading
HPC Performance Characterizatior
I/O
Input and Output
ACCELERATORS
GPU Offload
GPU Compute/Media Hotspots (preview)
CPU/FPGA Interaction
PLATFORM ANALYSES
System Overview
GPU Rendering (preview)
CPU/GPU Concurrency (deprecated)
Platform Profiler


Project Naviga...
sample (matrix)
r000hs
r001ue
r002ps
r003hpc
r004hpc-bad
r005hpc-bad
r006hpc-bad
r007hpc-bad
Welcome
r003hpc
HPC Performance Characterization
HPC Performance Characterization
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
KernelBFS
Grouping: Function / Call Stack
Function / Call Stack
CPU Time
Effective Time by Utilization
Idle Poor Ok Ideal Over
Spin Time
Overhead Time
Instructions R
[Loop@0x4021d0 in main. 21.842s 0s 0s 6,608,25
std::uniform_int_distributic 10.617s 0s 0s 34,915,59
[Loop@0x402820 in std::u 1.844s 0s 0s 4,846,05
[Loop@0x402770 in std::u 1.665s 0s 0s 1,986,48
[Loop@0x4027d8 in std::u 1.124s 0s 0s 2,912,97
[Loop@0x402198 in main. 1.046s 0s 0s 208,26
[Loop@0x401350 in main] 0.687s 0s 0s 40,05
clear_page_c 0.667s 0s 0s 32,04
_int_free 0.661s 0s 0s 1,607,34
call_function_interrupt 0.601s 0s 0s 173,55
func@0xffffffff817c930b 0.592s 0s 0s 77,43
copy_page_rep 0.379s 0s 0s 2,67
page_fault 0.331s 0s 0s 48,06
[Loop@0x-7ef9e2d0 in def 0.283s 0s 0s 48,06
Elapsed Time: 21.135s
CPI Rate: 2.174
Total Thread Count: 66
0s 2s 4s 6s 8s 9.484s 12s 14s 16s 18s 20s
Loading...
Thread
CPU Time
Thread
Effective Time
Spin and Overhea...
CPU Time
CPU Time
Spin and Overhea...
FILTER 100.0%
Process Any Process
Module Any Module
Call Stack Mode User functio
Loop Mode Loops and f
Inline Mode Show inline

OpenMP Region / Thread / Function / Call Stack,




Collection Log
Analysis Target
Analysis Type
Summary
Bottom-up
heart_de...
Source
Assembly
Assembly grouping: Address
So.
Source
CPU Time
Sourc
197 }
198 inline void init_send_bufs(const int rk4_id) {
199 // THIS HAS TO BE DONE SINGLETHREAD
200 for (auto &d : non_loc_dependees) {
201 int remote_rank = d.first;
202 std::vector<int> &ids_to_pack = d.second;
203 std::vector<double> &buf = non_loc_dependees_bufs[remote_rank];
204 for (int i=0; i<ids_to_pack.size(); i++) {
10.023ms
heart_demo
205 int j = ids_to_pack[i];
20.046ms
heart_demo
206 buf[i] = cnodes[j].state[DynamicalSystem::COUPLING_VAR_ID];
230.533ms
heart_demo
207 if (rk4_id == 3)
208 buf[i] += cnodes[j].rk4[2][DynamicalSystem::COUPLING_VA
209 else if (rk4_id > 0)
210 buf[i] += cnodes[j].rk4[rk4_id-1][DynamicalSystem::COUPI
180.417ms
heart_demo
211 }
212 }
213 }

Nuevo kernel: SpMV

```
std::cout << "[*] Calculating row structure...\n";
A.row_ptr.resize(rows + 1, 0);
for (int i = 0; i < rows; i++) {
    int nnz_in_row = std::max(1, poisson(gen));
    A.row_ptr[i + 1] = A.row_ptr[i] + nnz_in_row;
}
A.nnz = A.row_ptr[rows];
```

```
#pragma omp parallel for schedule(static) num_threads(NUM_THREADS)
for (int i = 0; i < rows; i++) {
    // Cada thread tiene su generador (evita contención)
    std::mt19937 local_gen(42 + omp_get_thread_num() * 1000 + i);
    std::uniform_int_distribution<> local_col_dist(0, cols - 1);
    std::uniform_real_distribution<> local_val_dist(0.0, 1.0);

    int start = A.row_ptr[i];
    int end = A.row_ptr[i + 1];

    for (int k = start; k < end; k++) {
        A.col_idx[k] = local_col_dist(local_gen);
        A.val[k] = local_val_dist(local_gen);
    }
}
```

```
void init_vector_numa_aware_simplified(std::vector<double>& v, int num_threads) {
    std::cout << "[*] Initializing vectors with NUMA-aware first-touch...\n";

    int n = v.size();
    #pragma omp parallel for schedule(static) num_threads(num_threads)
    for (int i = 0; i < n; i++) {
        v[i] = (double)(i + 1) / n;
    }
}
```

GRACIAS