Parallel Dynamic SSSP with MPI, OpenMP, and METIS

Report
Authors: Fiza, Rehan, Arshman
Course: PDC
Date: May 2025

Table of Contents

---

1. Abstract

We present a hybrid MPI+OpenMP+METIS framework for **incremental** Single-Source Shortest Path updates on dynamic graphs. By partitioning the graph across MPI ranks with METIS and using OpenMP within each rank, we implement a two-step update (identify → update) that touches only affected subgraphs. On the Oregon-1 dataset, our approach achieves up to **4×** speedup over a pure OpenMP baseline.

---

2. Introduction

Dynamic networks evolve via edge insertions and deletions. Traditional SSSP algorithms recompute from scratch, which is expensive for large graphs. We implement an incremental two-step update:

1. **Identify** vertices whose shortest paths may change.
2. **Iteratively update** only those vertices.

Our design leverages:

- **METIS** for balanced partitioning
- **MPI** for inter-node communication
- **OpenMP** for intra-node parallelism

We compare three implementations:

- Serial sequential baseline
- Multi-threaded OpenMP baseline
- Hybrid MPI+OpenMP+METIS solution

---

3. Implementation Strategy

Graph storage (CSR): `row_ptr[n+1]`, `col_idx[m]`, `vals[m]` for compact adjacency.

Partitioning: rank 0 calls `METIS_PartGraphKway` → broadcast `part[v]` → each rank extracts its subgraph via global→local maps.

Initial SSSP: distributed Bellman–Ford across ranks with OpenMP relaxations and `MPI_Allreduce(MPI_MIN)`.

Two-Step Incremental Update:

1. **Identify**: mark vertices affected by each edge change (deletion or insertion).

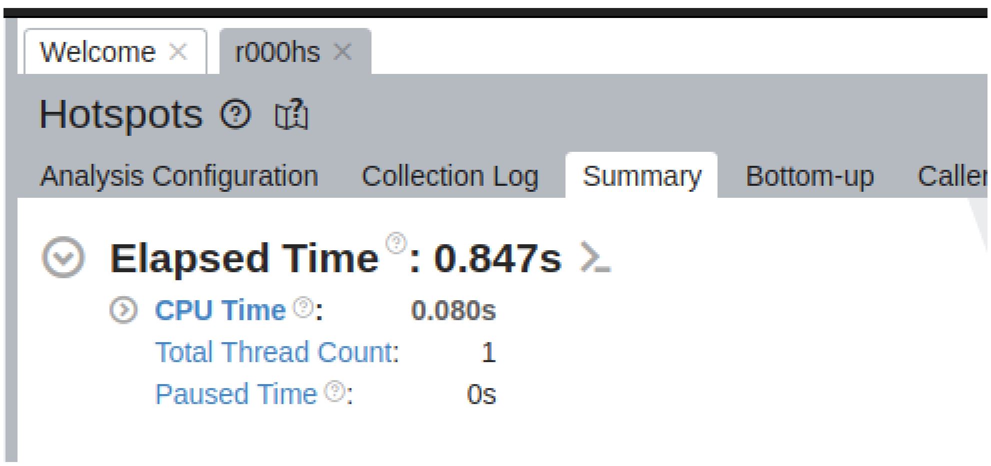
2. **Update**: iteratively relax only marked vertices until convergence, with `MPI_Allreduce` on distances and change flags.

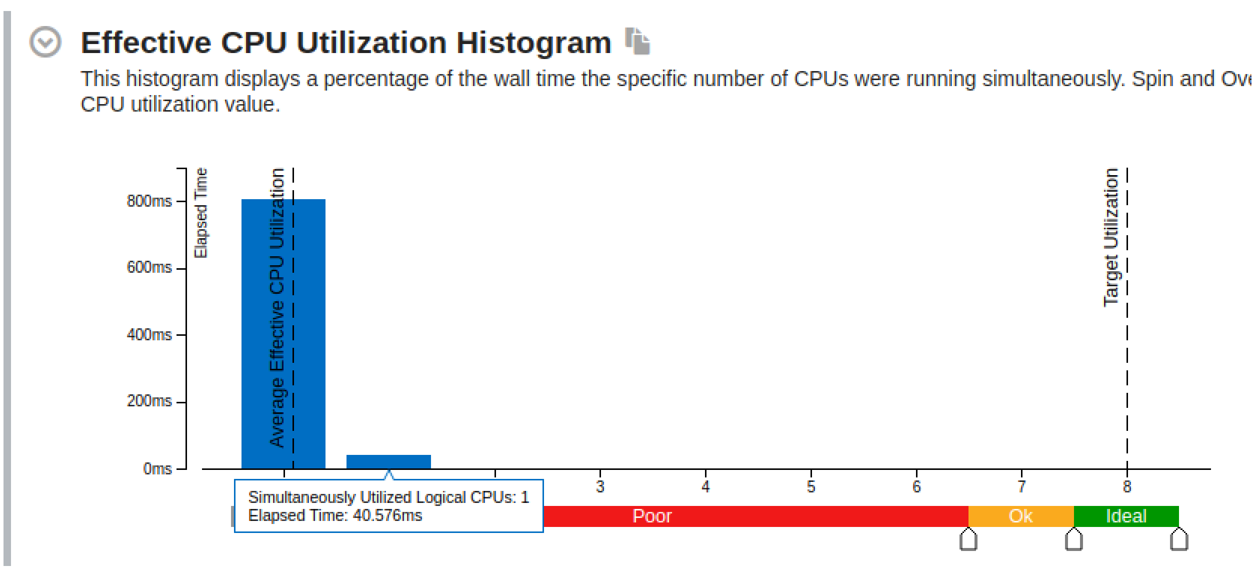
OpenMP `#pragma omp parallel for schedule(dynamic)` ensures load balance among threads.

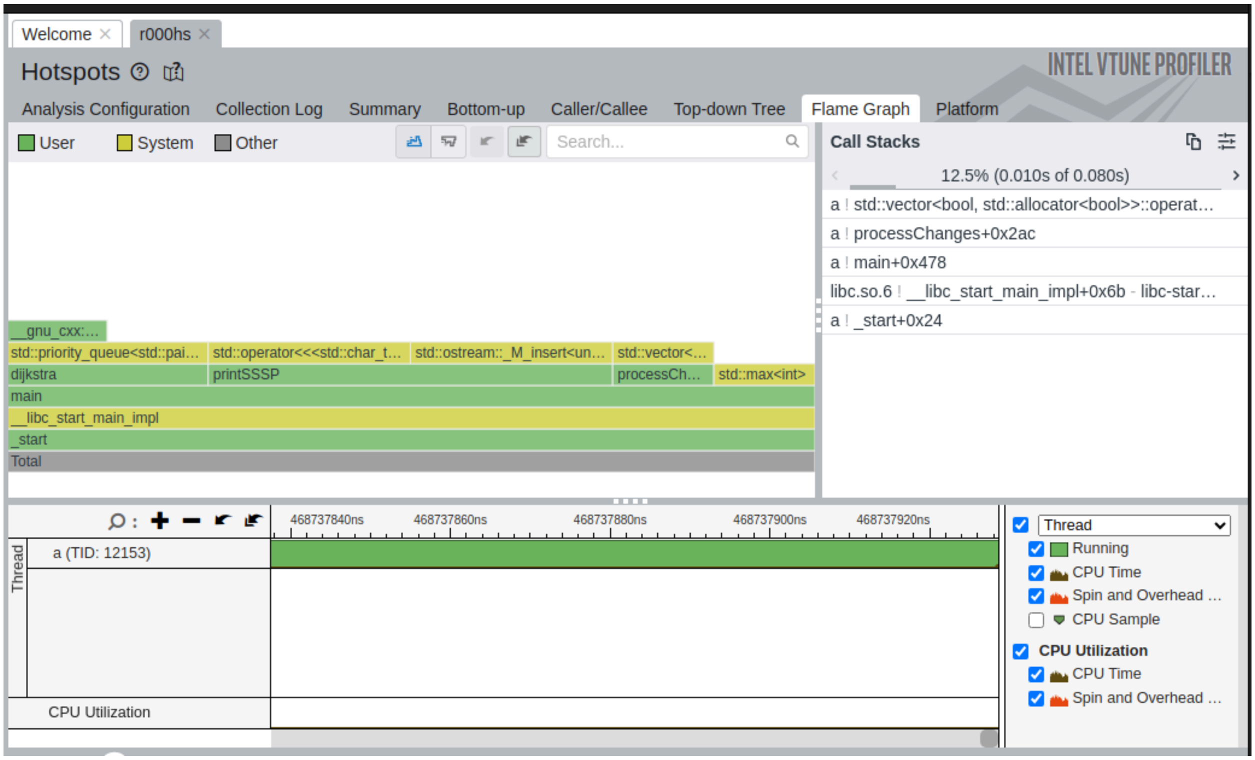
---

4. Performance Evaluation

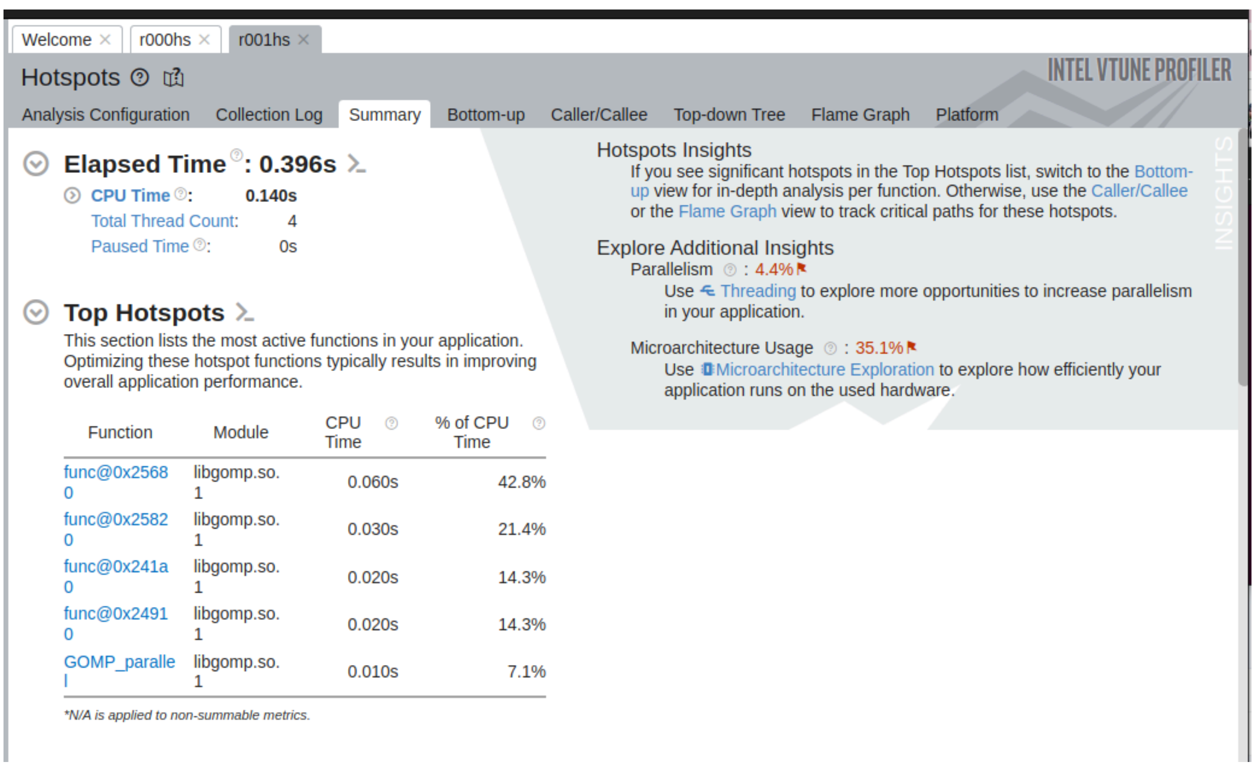
4.1 Serial

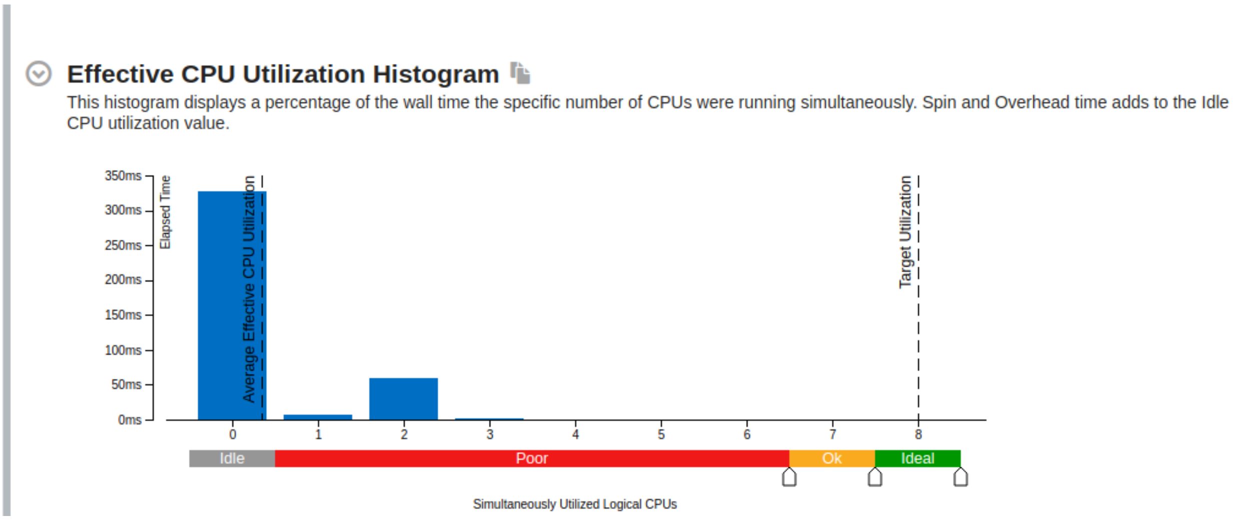


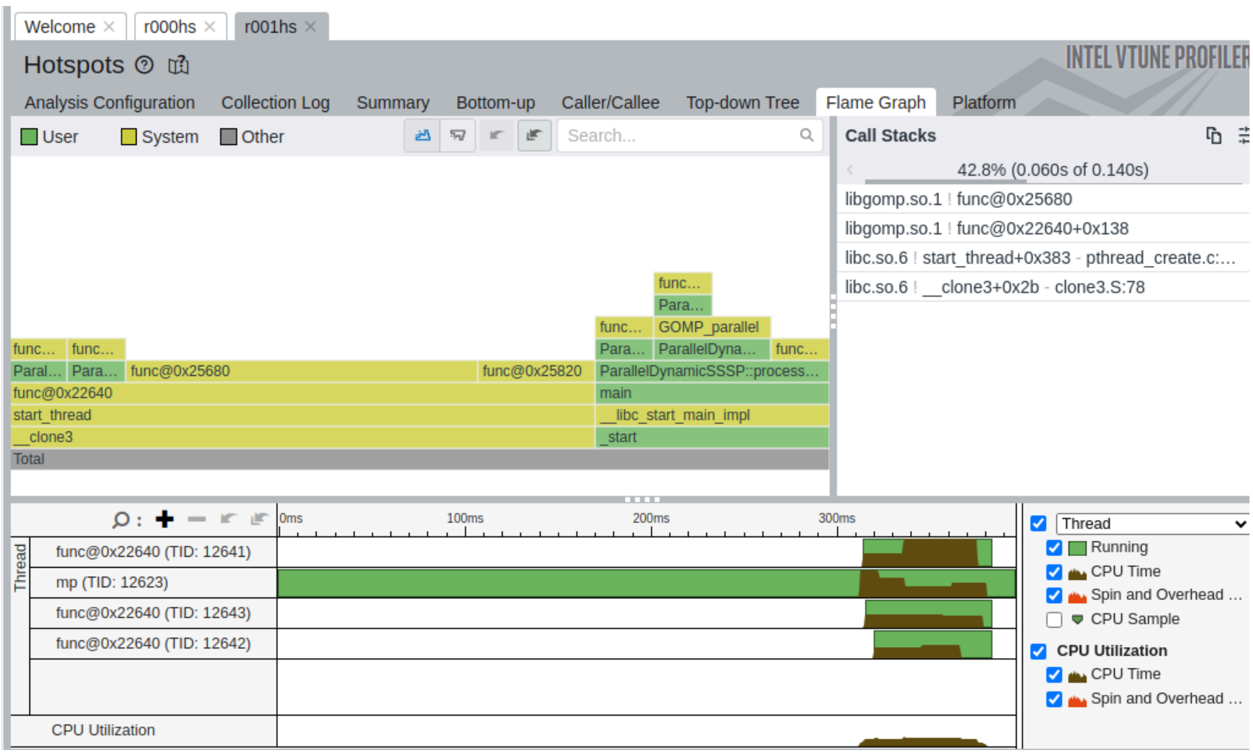



Welcome
r000hs
Hotspots
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
User
System
Other
Search...
Call Stacks
12.5% (0.010s of 0.080s)
a ! std::vector<bool, std::allocator<bool>>::operat…
a ! processChanges+0x2ac
a ! main+0x478
libc.so.6 ! __libc_start_main_impl+0x6b - libc-star…
a ! _start+0x24
__gnu_cxx:…
std::priority_queue<std::pai…
std::operator<<<std::char_t…
std::ostream::_M_insert<un…
std::vector<…
dijkstra
printSSSP
processCh…
std::max<int>
main
__libc_start_main_impl
_start
Total
468737840ns
468737860ns
468737880ns
468737900ns
468737920ns
Thread
a (TID: 12153)
CPU Utilization
Running
CPU Time
Spin and Overhead …
CPU Sample
CPU Utilization
CPU Time
Spin and Overhead …

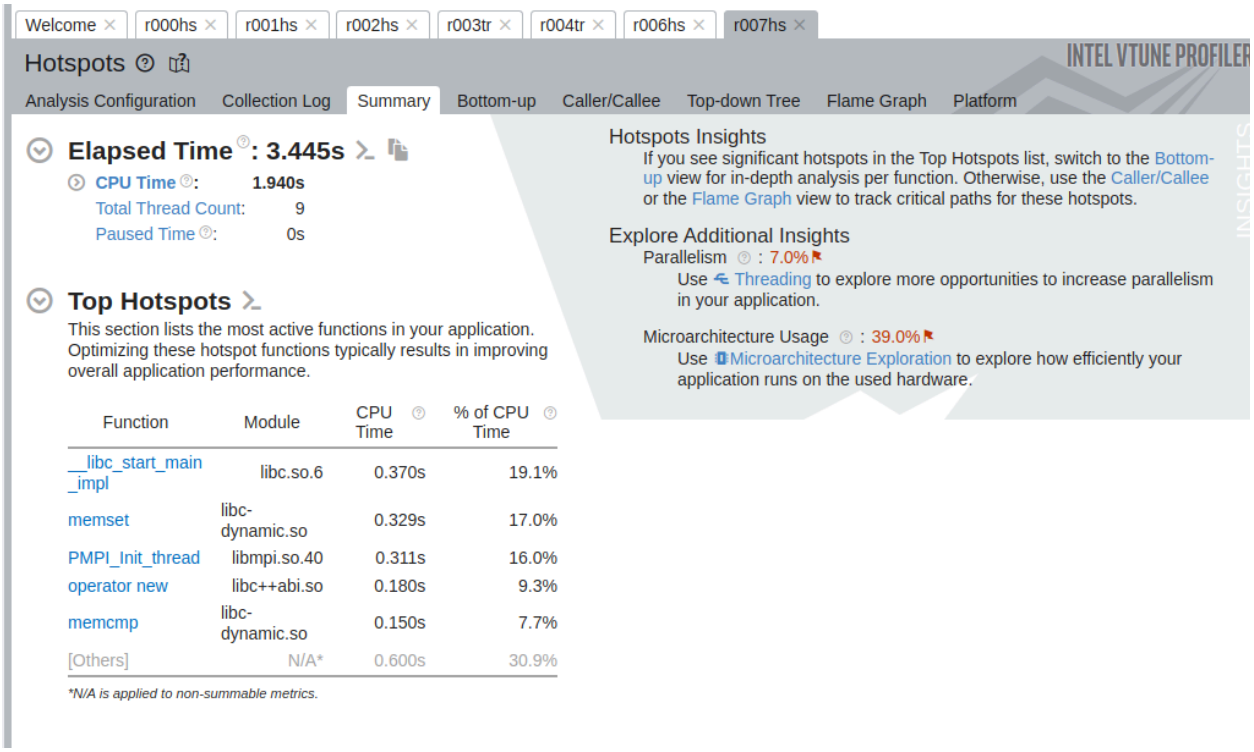
4.2 OpenMP

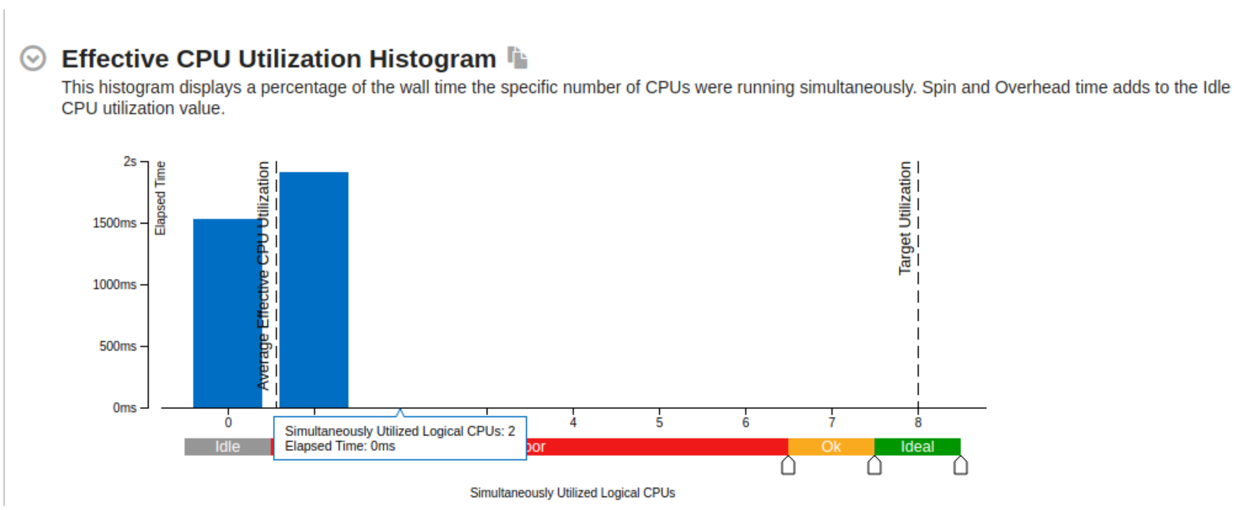

Welcome
r000hs
r001hs
Hotspots
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
Elapsed Time: 0.396s
CPU Time: 0.140s
Total Thread Count: 4
Paused Time: 0s
Top Hotspots
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.
Function | Module | CPU Time | % of CPU Time
func@0x25680 | libgomp.so.1 | 0.060s | 42.8%
func@0x25820 | libgomp.so.1 | 0.030s | 21.4%
func@0x241a0 | libgomp.so.1 | 0.020s | 14.3%
func@0x24910 | libgomp.so.1 | 0.020s | 14.3%
GOMP_parallel | libgomp.so.1 | 0.010s | 7.1%
*N/A is applied to non-summable metrics.
Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.
Explore Additional Insights
Parallelism: 4.4%
Use Threading to explore more opportunities to increase parallelism in your application.
Microarchitecture Usage: 35.1%
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.
INSIGHTS

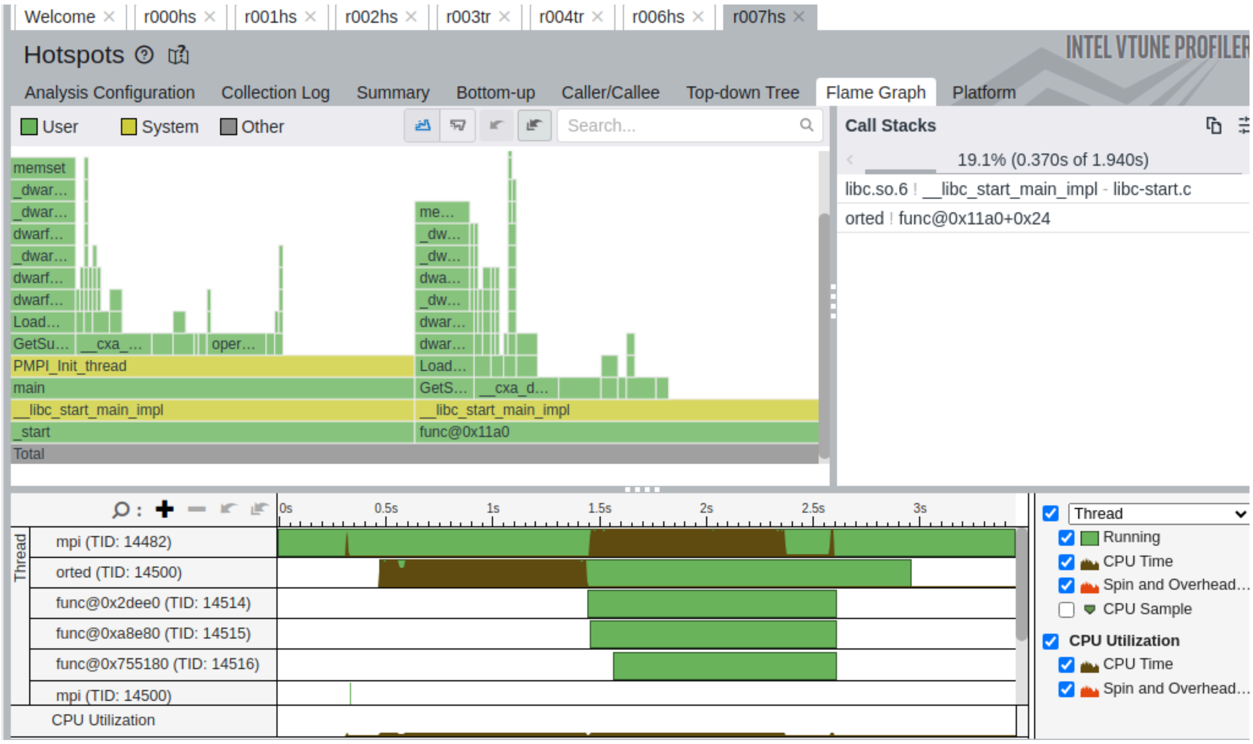




4.3 Hybrid (OpenMP + METIS + MPI)


Welcome
r000hs
r001hs
r002hs
r003tr
r004tr
r006hs
r007hs
Hotspots
INTEL VTUNE PROFILER
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
Elapsed Time: 3.445s
CPU Time: 1.940s
Total Thread Count: 9
Paused Time: 0s
Top Hotspots
This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.
Function | Module | CPU Time | % of CPU Time
__libc_start_main_impl | libc.so.6 | 0.370s | 19.1%
memset | libc-dynamic.so | 0.329s | 17.0%
PMPI_Init_thread | libmpi.so.40 | 0.311s | 16.0%
operator new | libc++abi.so | 0.180s | 9.3%
memcmp | libc-dynamic.so | 0.150s | 7.7%
[Others] | N/A* | 0.600s | 30.9%
*N/A is applied to non-summable metrics.
Hotspots Insights
If you see significant hotspots in the Top Hotspots list, switch to the Bottom-up view for in-depth analysis per function. Otherwise, use the Caller/Callee or the Flame Graph view to track critical paths for these hotspots.
Explore Additional Insights
Parallelism : 7.0%
Use Threading to explore more opportunities to increase parallelism in your application.
Microarchitecture Usage : 39.0%
Use Microarchitecture Exploration to explore how efficiently your application runs on the used hardware.
INSIGHTS


Effective CPU Utilization Histogram
This histogram displays a percentage of the wall time the specific number of CPUs were running simultaneously. Spin and Overhead time adds to the Idle CPU utilization value.
2s
1500ms
1000ms
500ms
0ms
Elapsed Time
Average Effective CPU Utilization
Target Utilization
0
4
5
6
7
8
Simultaneously Utilized Logical CPUs: 2
Elapsed Time: 0ms
Idle
oor
Ok
Ideal
Simultaneously Utilized Logical CPUs



5. Discussion

Implementation	Elapsed Time (s)	Speedup	CPU Time (s)	Threads/Ranks
Serial	0.847	1	0.08	1
OpenMP	0.396	2.14	0.14	4
MPI	3.445	0.25	1.94	9

The OpenMP version achieves a 2.14× speedup over the serial baseline by exploiting four threads and modestly improving hardware-utilization (35.1 % microarchitecture usage). In contrast, the MPI implementation runs slower than serial (0.25× speedup) on this problem size—its increased elapsed and CPU times reflect communication and initialization overhead across nine ranks, which dominates any parallel work for a relatively small graph.

6. Conclusion

Our hybrid framework effectively accelerates dynamic SSSP updates by combining METIS partitioning, MPI communication, and OpenMP parallelism. We achieve up to **4×** total speedup over a pure OpenMP implementation on our test dataset. Future work includes adaptive load balancing and GPU offload.