Using an Adaptive Mesh Refinement proxy code to assess dynamic load balancing capabilities for exascale

Rob Van der Wijngaart

Intel Labs, Intel Federal

Notices

Intel and Xeon are trademarks of Intel Corporation in the U.S. and/or other countries.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to http://www.intel.com/performance.

Acknowledgement This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facilitysupported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231

Agenda

- Background Parallel Research Kernels (PRK) Suite
- Motivation Adaptive Mesh Refinement (AMR) kernel
- AMR PRK specification
- Reference implementations
- Experimental results
  - Shared memory
  - Distributed memory
- Conclusions and future work

Background Parallel Research Kernels

Create test suite to study behavior of parallel systems

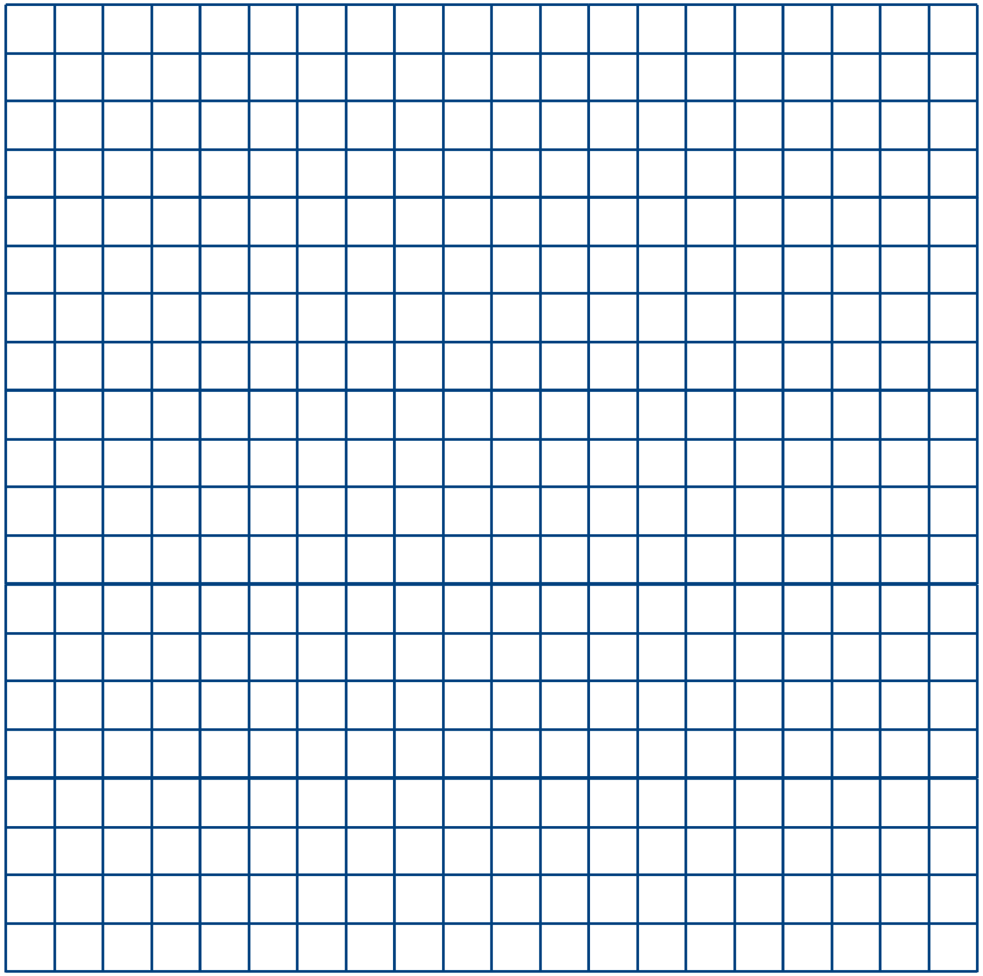
- Cover broad range of patterns found in real parallel applications
- Provide paper-and-pencil specification and generic reference implementations
- Keep kernels simple functionally
  - Easy porting to new runtimes/languages
  - Easy to understand by different domain scientists
  - Dominated by single feature, so convenient performance building block
- Parameterize kernels (problem size, iterations, # cores etc.)
- Make sure each kernel does actual work
- Include automatic verification test (analytical solution)
- Ensure enough expoitable concurrency (can be load balanced)
  - Make trivially statically load balanced

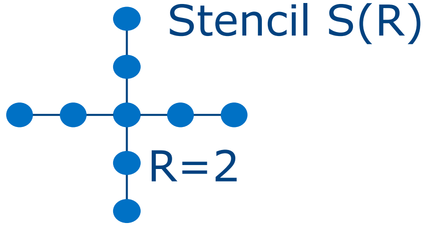
Motivation Adaptive Mesh Refinement (AMR) kernel

- However, exascale will require dynamic load balancing for mature workloads + system/network fluctuations
- Goal: Design and implement new kernels that:
  - Require dynamic load balancing at all system scales (algorithmic source)
  - Allow control of amount and frequency of workload adaptation
  - Have data dependencies, so load-balancing is non-trivial; improving load-balance usually increases communication
- Usage: Research vehicle to stress dynamic load-balancing capabilities of parallel runtimes + application frameworks
- Particle-in-Cell (PIC) PRK, IPDPS 2016: continually evolving mismatch between dependent data structures, fixed total work
- Adaptive Mesh Refinement (AMR) PRK, ISC 2017: abrupt, local variations in computational load (proxy for system disturbances), sudden increase/decrease in total work

AMR PRK Specification

Stencil PRK with Background Grid (BG) & periodic Refinement Grids (RGs)



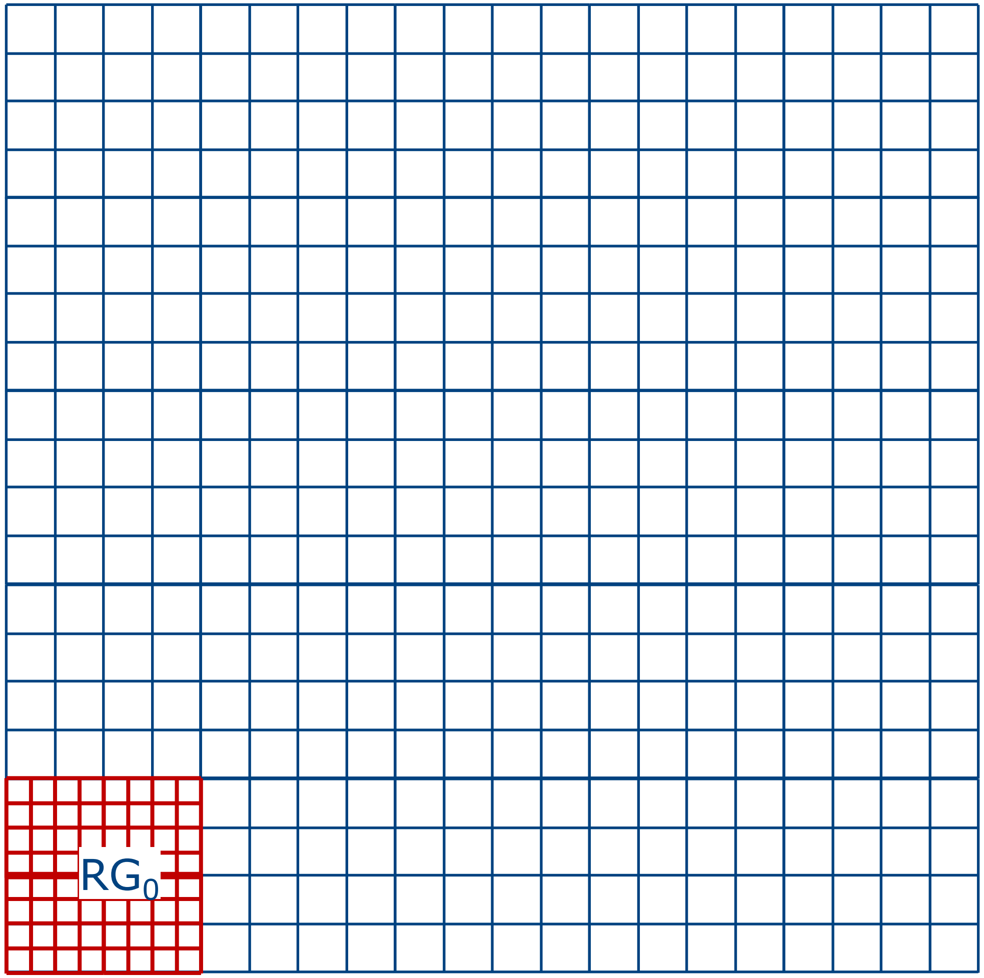


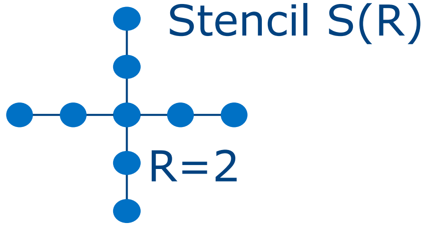
Parameters:

- Size of BG
- Size + refinement level of RGs
- Frequency + duration of refinement
- Iterations on RGs

AMR PRK Specification

Stencil PRK with Background Grid (BG) & periodic Refinement Grids (RGs)



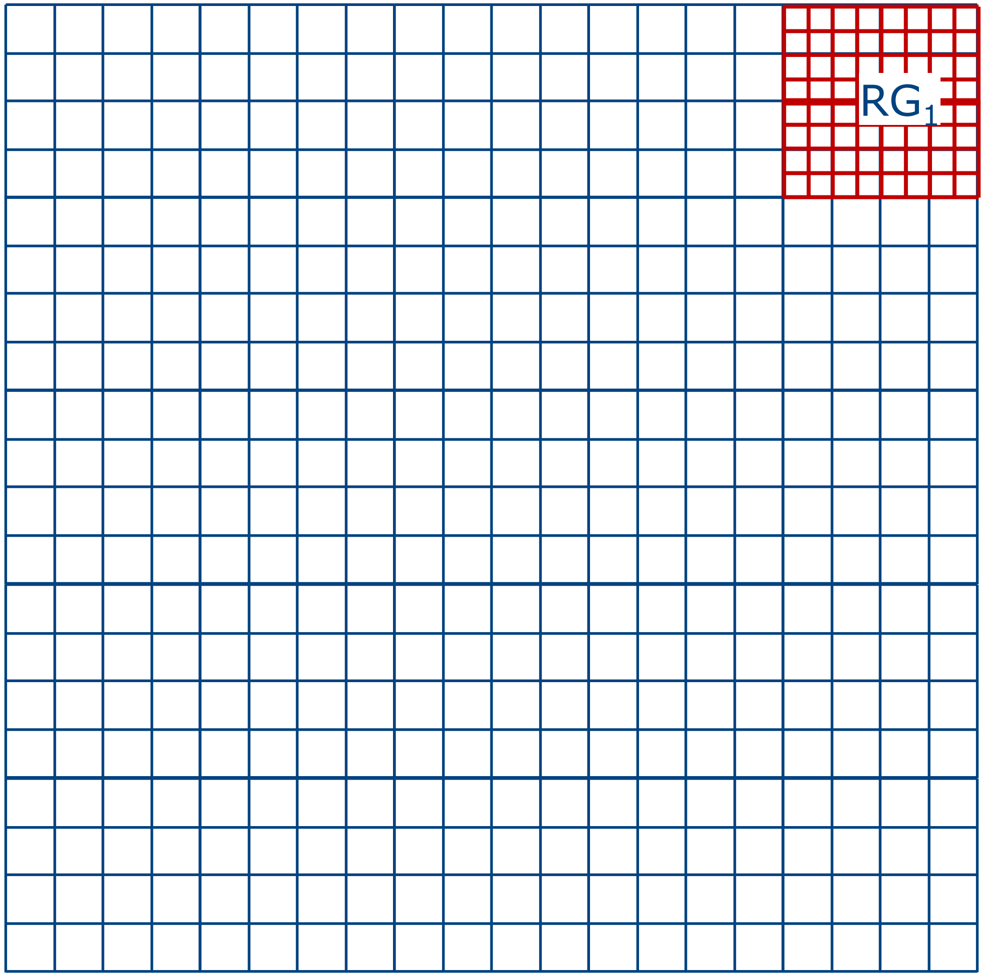


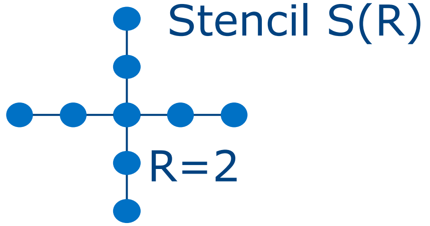
Parameters:

- Size of BG
- Size + refinement level of RGs
- Frequency + duration of refinement
- Iterations on RGs

AMR PRK Specification

Stencil PRK with Background Grid (BG) & periodic Refinement Grids (RGs)



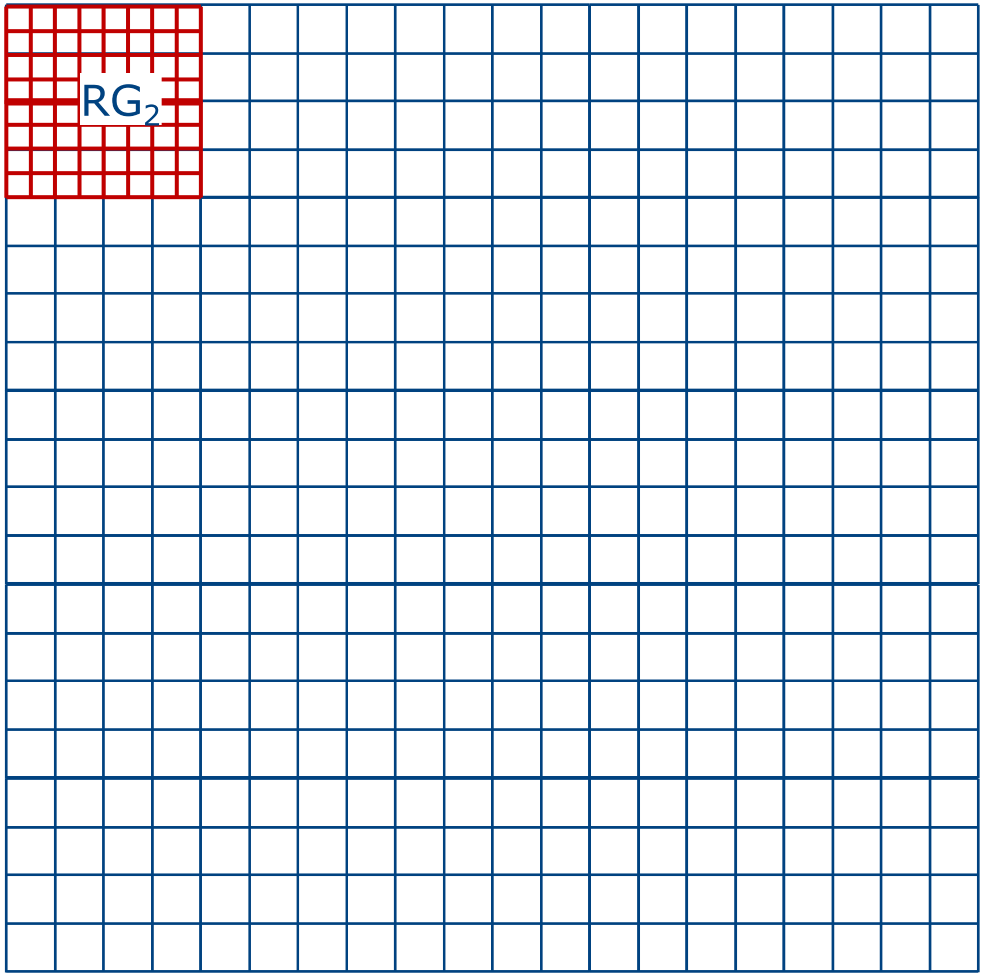


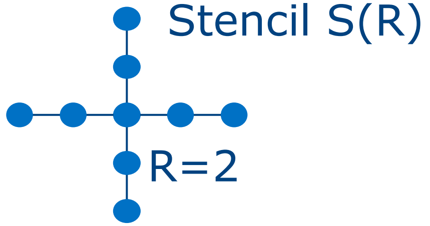
Parameters:

- Size of BG
- Size + refinement level of RGs
- Frequency + duration of refinement
- Iterations on RGs

AMR PRK Specification

Stencil PRK with Background Grid (BG) & periodic Refinement Grids (RGs)



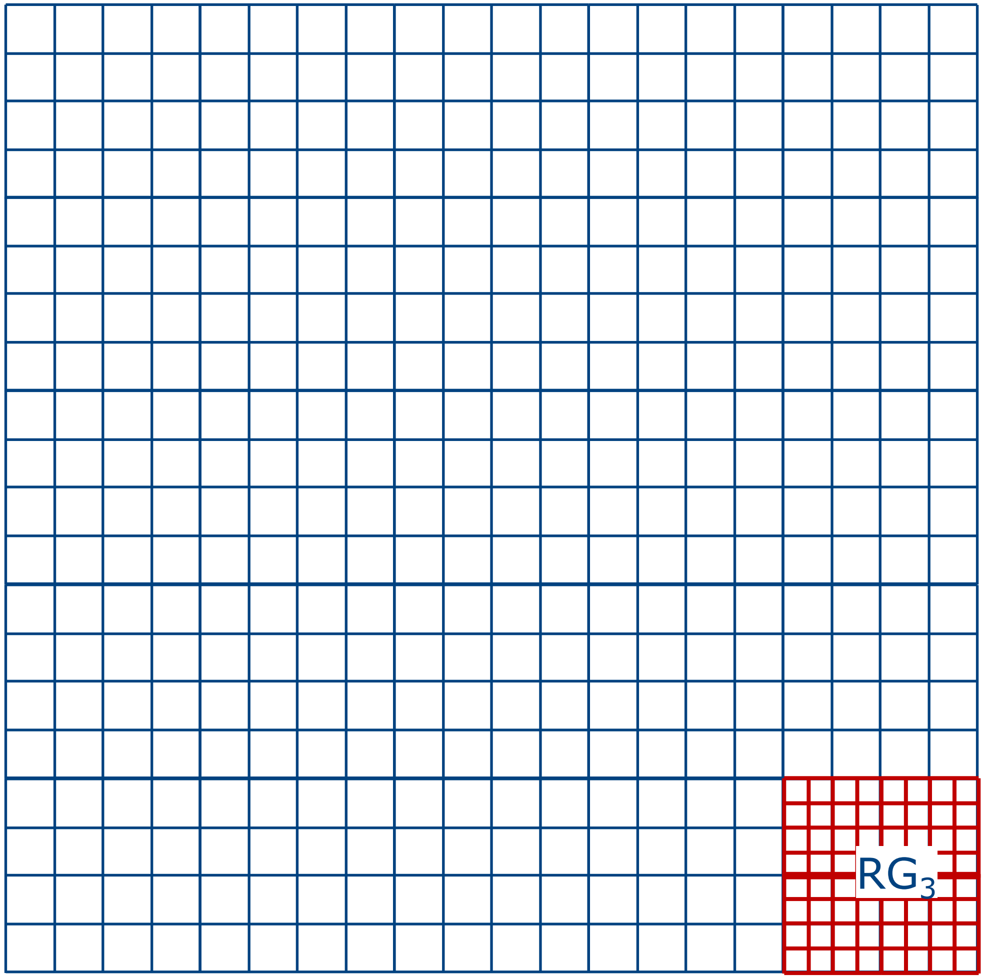


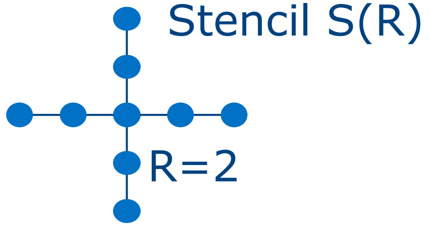
Parameters:

- Size of BG
- Size + refinement level of RGs
- Frequency + duration of refinement
- Iterations on RGs

AMR PRK Specification

Stencil PRK with Background Grid (BG) & periodic Refinement Grids (RGs)





Parameters:

- Size of BG
- Size + refinement level of RGs
- Frequency + duration of refinement
- Iterations on RGs

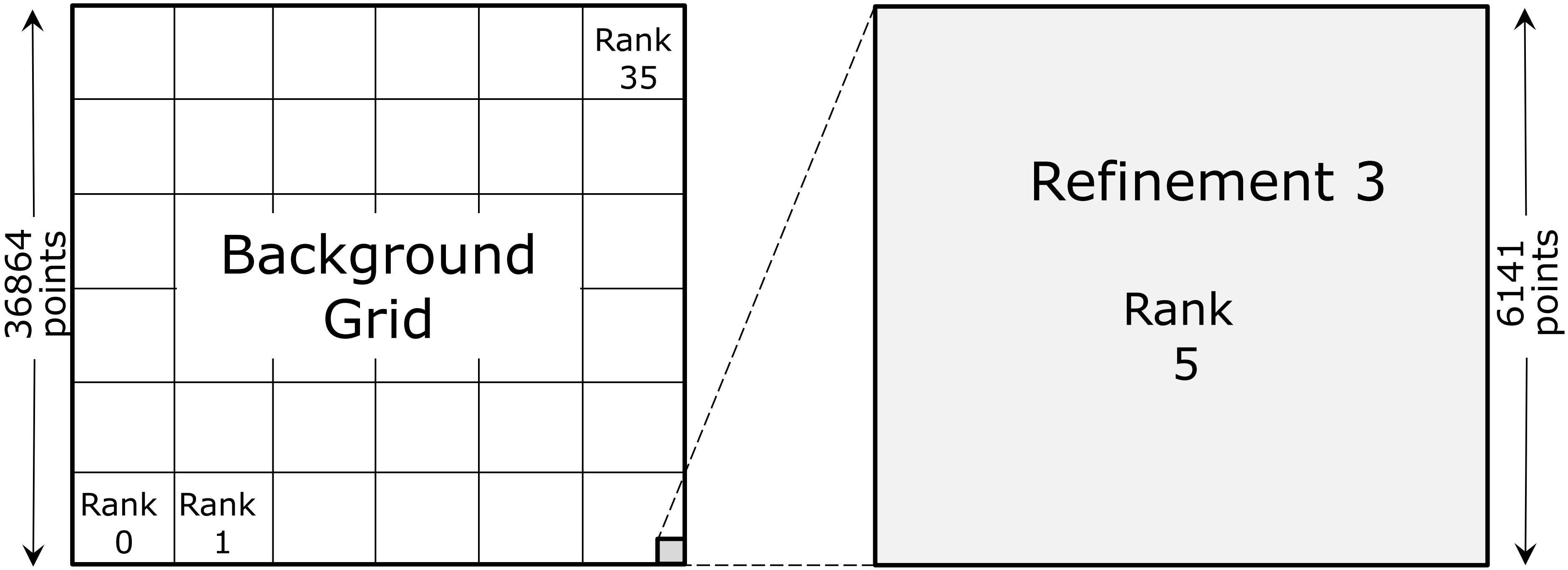
Reference implementations

- Application level "dynamic load balancing" (MPI)
  - No over-decomposition
  - When refinement appears:
    - FINE-GRAIN: Divide RG work evenly among all ranks
    - HIGH-WATER: Divide RG $\cup$ BG evenly among all ranks
    - NO-TALK: Assign RG work to rank(s) owning corresponding part(s) of BG
- Runtime orchestrated dynamic load balancing (Adaptive MPI)
  - Relies on canonical MPI partitioning (above), with over-decomposition

Experiments

- **Shared memory**: Intel® Xeon® E5-2699v3, 2.30 GHz, 64 GB memory, 2x18 cores (full occupation)
- **Distributed memory**: NERSC Edison, Cray XC30, Intel® Xeon® E5-2695v3, 2.40 GHz, 64 GB memory, 2x12 cores (full occupation)
- SMP experiment: NO_TALK, BG= 36864^2, RG=1536^2 (2-level refinement → 6141^2 points), 400 time steps, 1 RG iter/BG iter, RG Duration = {10,20,40} time steps, Period = 2*Duration Implications:
  - RG coincides with single BG patch, even with over-decomposition
  - RG size = BG patch size
  - #iters with refinements = #iters without refinements
- Adaptive MPI (AMPI): Over-decomposition = {2,4,8}, LB={refine,greedy}, migration delay = 1-5 time steps, use isomalloc to migrate ranks

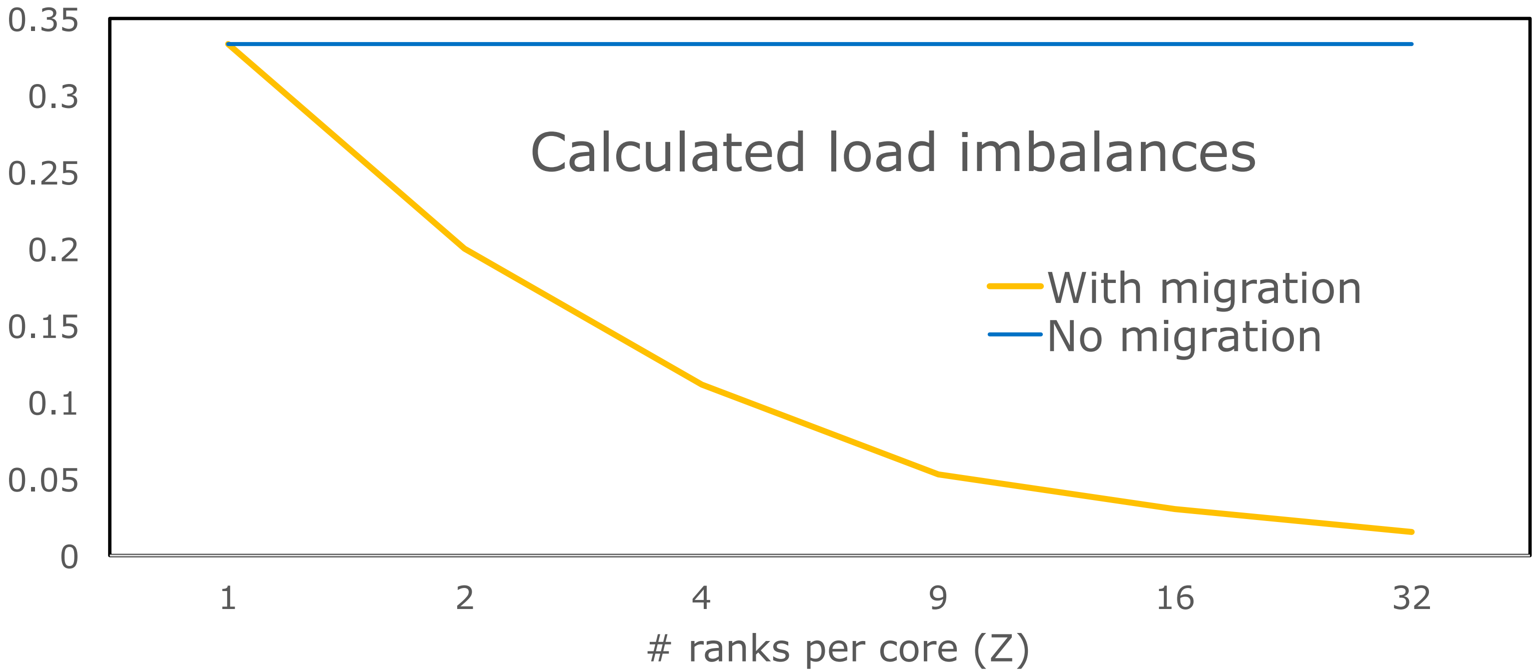
Experimental grid configuration



Work assignment policy NO-TALK

Experimental results, shared memory

- Theory (load imbalance = $1\text{-}T_{avg}/T_{max}$):
  - If over-decomposed (Z ranks per core) but no rank migration allowed (equivalent to plain MPI), load imbalance = 1/3
  - If rank migration allowed (optimum if core with RG rank moves off all ranks with only BG tiles), load imbalance = 1/(2Z+1)



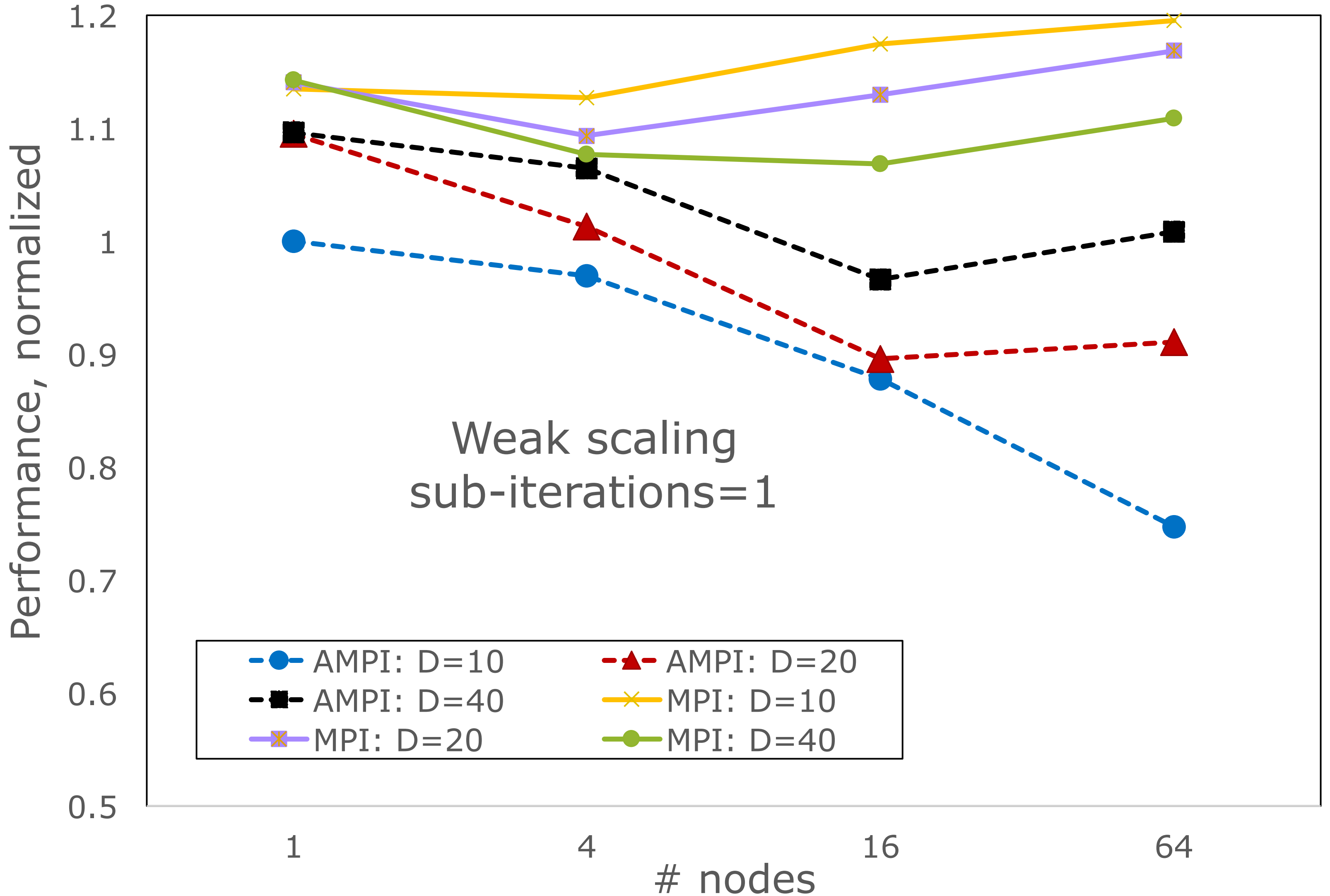
Experimental results, shared memory

- Observations:
  - LB=Refine: plain MPI and AMPI perform the same for all parameters: 41.1 GFlops/s ±2.7% (~5% migrate)
  - LB=Greedy: 35.5 Gflops ±5.3% (~100% migrate)
  - AMPI performance independent of "noise" frequency, migration delay, degree of over-decomposition
  - #ranks migrating irregular, despite regular disturbances
  - Plain Stencil PRK iteration times for RG on 1 rank and BG on 36 ranks 0.14s and 0.58s, respectively
  - If increasing work on RGs by 4x and 16x while keeping BG work unchanged, again AMPI perf ≈ plain MPI perf
  - If reducing RG and BG work by 16x (noise Hz 16x), AMPI perf for durations 20 & 40 ≈ plain MPI perf, but AMPI perf for duration 10 down 24%

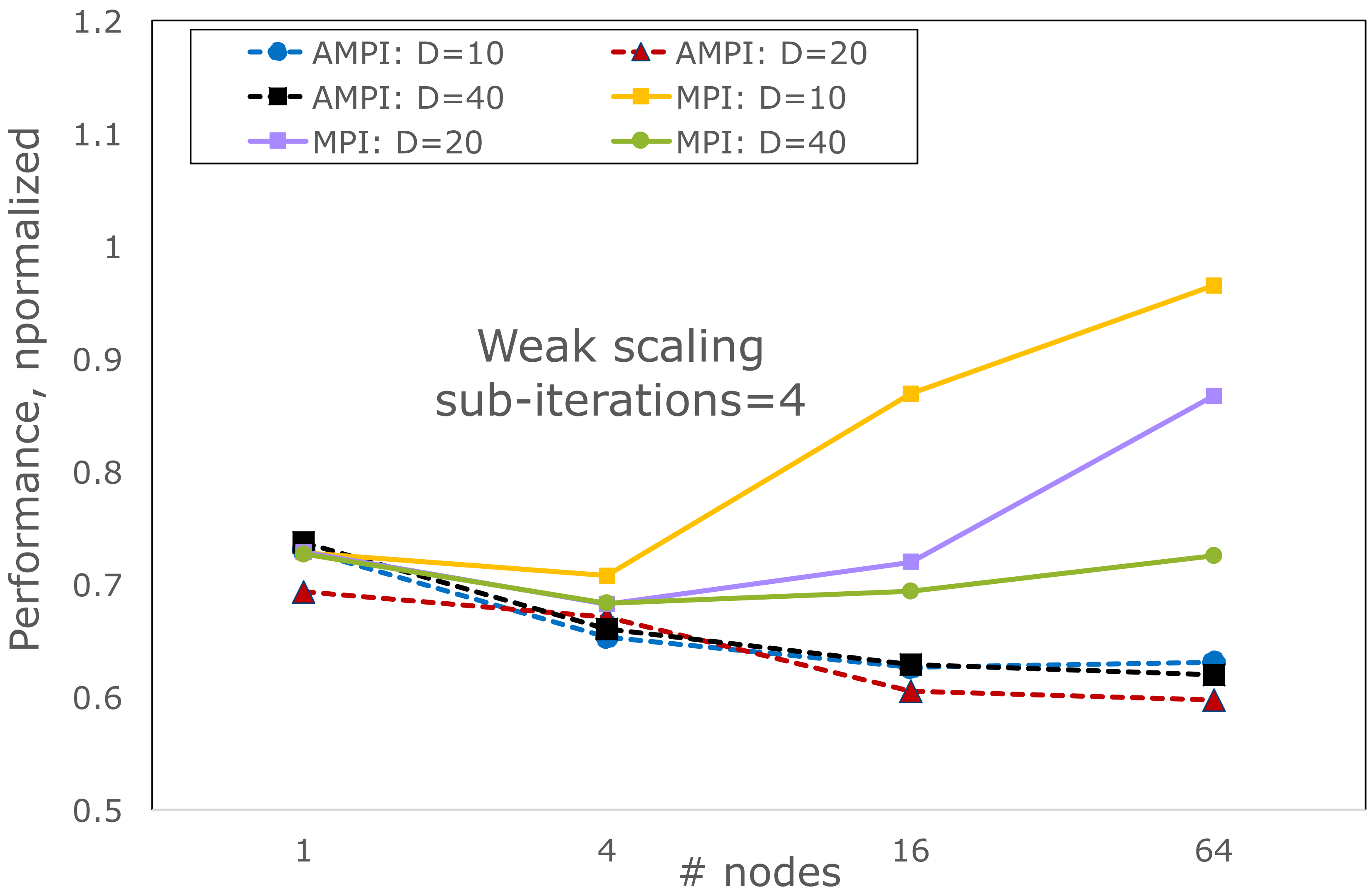
Experimental results, distributed memory

- Only used LB=Refine
- Weak scaling, so 4x number of nodes, BG grows by 2x in each coordinate direction
- RG size constant and same as in shared memory case: ratio of BG/RG work for rank receiving RG remains constant
- Fix overdecomposition at 4, migration delay at 2 iters
- Duration = {10,20,40}
- Use Pack/Unpack for rank migration
- First experiment: 1 RG iter/ BG iter (same as shared memory experiment).

Distributed memory results, 1 subiteration



Distributed memory results, 4 subiterations



Conclusions and future work

- Conclusions
  - AMR good, flexible proxy for localized disturbances
  - Adaptive MPI convenient vehicle for quick comparison with legacy runtime
  - Adaptive MPI implementation with dynamic load balancing does not manage to improve performance over non-adaptive MPI
- Future work
  - Repeat AMPI experiments with “oracle load balancer”
  - Test dynamic load balancing capabilities of other disruptive, task-based runtimes (Legion, OCR, HPX3/5) with AMR

intel®

Backup material

Specification details

Parameters

- T : total number of iterations (background grid)
- R: radius of difference stencil
- n: linear dimension of square background grid (n^2 points, mesh spacing is unity)
- r: refinement level (mesh size of refined grid is 2^{-r})
- k: linear dimension of refinement in terms of BG cells ($(k*2^r+1)^2$ points in each refinement)
- P : duration in terms of iterations on the BG of one full cycle of activation of one refinement until that of the next (*period*)
- D: duration in terms of iterations on the BG of activity on each refinement; $D \leq P$
- d: number of iterations on a refinement per iteration on the BG

Specification details

(Re-)initialization

- $In[0](x,y) = c_x x + c_y y$
- $In_i[t] = \phi\,(In[t])$, with ϕ bi-linear interpolation (exact for linear field)

Update

- Increase In and In_i by constant after each stencil application

Verification

- S is numerical equivalent of ∇ (exact for linear field): $\nabla(c_x x + c_y y + \text{const}) = c_x + c_y$
- Count number of iterations η_i on $g_i \rightarrow Out_i[T](x,y) \equiv \eta_i * (c_x + c_y)$
- $Out[T](x,y) = T * (c_x + c_y)$
- $In[t](x,y) = c_x x + c_y y + t$, so: $\underline{In[T](x,y)} = (c_x + c_y)(n-1)/2 + T$
- Count number of updates ν_i on g_i since last interpolation at time $\theta_i \rightarrow \underline{In_i[T](x,y)} \equiv (c_x + c_y) * k/2 + \nu_i + f(corner_i) + \theta_i$

$corner_i$ = coordinates of bottom left corner point of g_i

Three example AMR scenarios

1. n=1000, 10 workers, r=1, k=100, P=3, D=1, d=1. Refinement has 1% of work of BG, lasts 1 iteration, then waits for 2 iterations until next refinement. OK to add refinement work to worker covering same part of BG (~10% load imbalance)
2. n=1000, 100 workers, r=1, k=100, P=3, D=1, d=1. Not OK to add refinement work to worker covering same part of BG (100% load imbalance). Rapid (dis)appearance requires frequent load balancing
3. n=1000, 100 workers, r=4, k=6, P=30, D = 10, d = 5. Refinements ≈number of grid points as in scenario 1, but cover much smaller fraction of the BG; activated 10x slower than in that case, persist 50x longer, so automatic load balancing may respond effectively to changes in load