# Parallel Programming Models for Heterogeneous and Manycore Computing Nodes

Yonghong Yan[1], Barbara M. Chapman[1], Vivek Sarkar[2], and Bronis R. de Supinski[3]
[1]University of Houston, [2]Rice University, [3]Lawrence Livermore National Laboratory

## Introduction

The High Performance Computing (HPC) community is heading toward the era of exascale machines, expected to exhibit an unprecedented level of complexity and size. The community agrees that the biggest challenges to future application performance lie with efficient node-level execution. The amount of processing power within each node is expected to grow by three to four orders of magnitude through a significant increase in on-node concurrency and through innovations in diverse areas from packaging to memory hierarchies. These nodes might be comprised of many identical compute cores in multiple coherency domains, or they may contain specialized cores that perform a restricted set of operations with high efficiency, together making heterogeneity and manycore design a typical architecture. Although we anticipate physically shared memory within each node, access speeds will vary considerably between cores and between types of memory. Further, the node may present distinct address spaces to different computing elements, as demonstrated in today's accelerator architectures.

One of the critical challenges for using the massive parallelism capability within an application is the provision of programming models that facilitate the expression of the required levels of concurrency while permitting an efficient implementation by the system software stack. Node-level parallel models include thread-primitives such as the conventional pthreads and C++11 thread and Boost thread library for CPU/SMPs; low-level programming models for GPGPU accelerators such as CUDA and OpenCL; directive-based programming models for accelerators such as OpenMP and OpenACC while OpenMP also provide rich sets of features supporting general shared memory systems; and other less-widely used options for example Cilkplus, TBB and vector primitives. While internode-programming models can also be used for programming intranode parallelism, typically MPI, Charm++, PGAS (UPC, CAF and OpenSHMEM) and APGAS (X10 and Chapel), they are mainly designed for handling data movement across networks for internode parallelism.

## Features and models compared

A single intranode-programming model that meets the diverse requirements of exascale computations (both applications and architectures) must support the following features:
**Parallelism patterns:** The model should allow users to specify at least the following three parallelism patterns: data parallelism, which typically maps well to the manycore accelerators and vector architectures; asynchronous task parallelism that easily supports certain application algorithms for parallel computations; and data-driven computations.
**Data movement and locality control:** The model must support (and not merely enable) NUMA architectures, data-computation binding and explicit data mapping between different memory spaces.
**Synchronizations:** The programming model should support synchronization between various parallel work units, such as barrier operations, point-to-point synchronization and phase-based synchronization for streaming computations.
**Mutual exclusion:** Interfaces such as locks and hyberobjects are still widely used for protected data

access. Architectural changes such as transactional memory provide alternatives to achieve similar data protection, which could also be part of the interfaces of a parallel model.

**Interoperability:** One intranode-programming model should support interoperability with other models within a node and also with the internode-programming model and libraries such as MPI and PGAS.

**Other important features:** Specifically, I/O, error/resilience, tools support, and language/library binding are also important features that a programming model should support.

With these features, we have compared a list of node-level programming model in Figure 1.

| | **Data Parallelism** | **Async task Parallelism** | **Data-Driven computation** | **Data movement and locality control** | **Synchronization** | **Mutual Exclusion** | **Interop** | **Language binding or lib** | **Resilience** | **Tool support** |
|---|---|---|---|---|---|---|---|---|---|---|
| **OpenMP (4.0)** | parallel for, SIMD, teams and distribute | task/taskwait | depend (in\|out\|inout) | affinity and places | Barriers and reductions | Locks, critical, atomic | 5.0 (?) | language extension(C/C++, Fortran), pragma | omp cancel | OMPT |
| **Cilkplus** | cilk_for/SIMD | cilk_spawn/sync | None | None | hyperobjects | Locks | With TBB (?) | language extension, C/C++ | abort(?) | Cilkcreen Cilkviewer |
| **Habanero** | foreach | async/finish | data-driven future | places | phaser | isolation | HCMPI | C and Java | None | None |
| **pthread** | None | pthread_create/join | None | None | pthread_cond | mutex/lock | N/A | C/C++ lib | pthread_cancel | None |
| **qthread** | qthread_for | qthread_fork | qthread_readFE, future_init | shepherd | reduction | lock | N/A | C/C++ lib | ? | ? |
| **Intel TBB** | parallel_for/while/do, etc | Generic algs, tasks | None | None | barrier, reductions, etc | mutex | With Cilkplus (?) | C/C++ lib | ? | None |
| **MS .Net Parallel Extensions** | PLINQ, parallel.For/foreach | TPL | None | None | Futures | Locks, monitors | ? | C# | None | ? |
| **OpenACC** | acc | None | None | data | barrier, reduction | No | No | C/C++ and Fortran | None | None |
| **CUDA 5.0** | SIMT | None | None | host-device lib | Barriers | None | N/A | C/C++ | None | memcheck, gdb |
| **OpenCL** | SIMD | None | None | host-device lib | Barrier | No | N/A | C/C++ | None | None |
| **Java Concurrency** | thread pool | Executors, Task Queues | None | None | Synchronizers | Locks, monitors, atomic classes | ? | Java | ? | ? |

Figure 1: Features compared of intranode parallel programming model

## Conclusion

As shown in Figure 1, OpenMP clearly supports more features than others in the categories that we discussed in the last section. Known as a productive parallel programming model for shared memory machines, OpenMP has evolved in the latest 4.0 and the ongoing 5.0 developments into a multi-language and high-level programming standard to address the needs for a wider community than HPC.

The choices for a parallel model for a particular applications and/or hardware architecture depends on both the programmability of the model and the performance delivered to users by the implementations, and the tradeoff between them. For example, GPGPU accelerator support in high-level programming interface, one of the urgently needed features of parallel node-level programming, is now available in both OpenACC and OpenMP, with OpenACC being developed earlier and with more existing compiler support. However, a wide variety of users would still like to use proprietary CUDA model or NVIDIA libraries for their applications despite their productivity challenges because they deliver higher performance than the high-level programming models. The selection of a programming model needs experimental studies and profiling, as well as the evaluation of programming model implementations, combining the analysis of application algorithms, data layout and access patterns with regards to the hardware architectures.