Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement

Yonghong Yan, Jisheng Zhao, Yi Guo, and Vivek Sarkar

Department of Computer Science, Rice University
{yanyh,jisheng.zhao,yguo,vsarkar}@rice.edu

Abstract. Modern computer systems feature multiple homogeneous or heterogeneous computing units with deep memory hierarchies, and expect a high degree of thread-level parallelism from the software. Exploitation of data locality is critical to achieving scalable parallelism, but adds a significant dimension of complexity to performance optimization of parallel programs. This is especially true for programming models where locality is implicit and opaque to programmers. In this paper, we introduce the *hierarchical place tree* (HPT) model as a portable abstraction for task parallelism and data movement. The HPT model supports co-allocation of data and computation at multiple levels of a memory hierarchy. It can be viewed as a generalization of concepts from the Sequoia and X10 programming models, resulting in capabilities that are not supported by either. Compared to Sequoia, HPT supports three kinds of data movement in a memory hierarchy rather than just explicit data transfer between adjacent levels, as well as dynamic task scheduling rather than static task assignment. Compared to X10, HPT provides a hierarchical notion of places for both computation and data mapping. We describe our work-in-progress on implementing the HPT model in the Habanero-Java (HJ) compiler and runtime system. Preliminary results on general-purpose multicore processors and GPU accelerators indicate that the HPT model can be a promising portable abstraction for future multicore processors.

1 Introduction

Modern computer systems feature deep memory hierarchies, multiple heterogeneous or homogeneous computing units, and an emphasis on thread-level parallelism instead of instruction-level parallelism. Exploitation of data locality is critical to achieving scalable parallelism, but it adds a significant dimension of complexity to performance optimization of parallel programs.

In this paper, we introduce a *hierarchical place trees* (HPT) abstraction that supports co-location of data and computation at multiple levels of a memory hierarchy, as well as three kinds of data movement — implicit access, explicit in-out parameters, and explicit asynchronous transfer — to support different communication mechanisms across memory hierarchies. HPT can be viewed as a generalization of concepts from the Sequoia and X10 programming models, resulting in capabilities that are not supported by either. Compared to Sequoia [5], HPT allows for the three kinds of data movement listed above, rather than just

explicit data transfer between adjacent levels; and for dynamic task scheduling rather than static task assignment. Compared to X10 [7], HPT provides a hierarchical notion of "places" for both computation and data assignment, as well as implicit data access across places.

We use a prototype implementation of HPT model to demonstrate its use as a portable interface for homogeneous (CPU) and heterogeneous (GPU) multicore parallel systems. The evaluation results show performance improvement for locality-sensitive applications when they run with HPT configurations that are consistent with the memory hierarchy of the target systems. The magnitude of the improvement is expected to increase in future many-core systems, where locality and data movement will be even more critical for performance than today.

The rest of the paper is organized as follows. Section 2 summarizes past work on X10's places and the Sequoia system. Section 3 introduces the Hierarchical Place Trees (HPT) model. Section 4 presents the programming interfaces for using HPT in the Habanero-Java (HJ) programming language, compiler and runtime. Though this paper discusses HPT in the context of HJ, HPT should be applicable to other parallel programming models as well. Section 5 presents our preliminary experimental results. Finally, Section 6 discusses related work and Section 7 contains our conclusions.

2 Background

This section summarizes the place and activity features of X10 concurrency and data distribution models, and the Sequoia's approach to support portable application development across machines of different memory hierarchies.

2.1 Places and Activities in X10

X10 is an Asynchronous Partitioned Global Address Space (APGAS) language featuring task parallelism and locality control through the use of *places* [7, 9]. A place in X10 (Chapel uses the term, *locale*, for a similar concept [1]) enables co-location of asynchronous tasks and shared mutable locations. A computation unit in a place is called an *activity* (task), which denotes a dynamic execution instance of a piece of code acting on application data. X10 places are used to support both data distributions and computation distributions. Place values can be obtained in the following ways: (i) *here* is a constant that denotes the place where an activity is executing; (ii) given an object reference V, $V.location$ gives the reference of the place where the object resides; (iii) a distribution is a map from indices to places that describes how a distributed array is laid out. Given a distribution d, $d[p_i]$ gives the place where the distribution maps the index p_i.

An X10 program execution contains multiple places; the same X10 program runs, unmodified, regardless of the number of places supplied by the system. Application data may be distributed among places using defined distribution policies. In pure X10, all data accesses must be place-local, and a remote data access can only be performed by explicitly creating a new activity at the remote place which serves the explicit asynchronous data transfer. In the HPT model

presented in this paper, we permit implicit access to remote locations in addition to explicit data transfers.

X10 provides the async and finish constructs for specifying concurrent activity creation and synchronizations. The async statement, $async\ (\langle place \rangle)\ \langle stmt \rangle$, causes the parent activity to fork a new child activity that executes $\langle stmt \rangle$ at $\langle place \rangle$. Execution of the async statement returns immediately, i.e., the parent activity can proceed immediately to its following statement. The finish statement, $finish\ \langle stmt \rangle$, causes the parent activity to execute $\langle stmt \rangle$ and then wait until all the activities created within $\langle stmt \rangle$ have terminated (including transitively spawned activities).

The statement *foreach (point p : R) S* supports parallel iteration over all the points in region R by launching each iteration as a separate async in the same place as the parent activity. A *point* is an element of an n-dimensional Cartesian space ($n \geq 1$) with integer-valued coordinates. A *region* is a set of points that specifies the iteration space for the foreach statement. Likewise, the statement *ateach (point p : D) S* supports parallel iteration over each point p in distribution D by launching an async for point p at place $D[p]$. Neither the foreach nor ateach statement has an implicit finish (join) operation. But its termination can be ensured by enclosing it within a finish statement at an appropriate outer level.

2.2 Parallel Memory Hierarchies in Sequoia

The Sequoia programming language and runtime [5] were designed to facilitate the development of portable applications across machines with different memory hierarchies. In Sequoia, a memory hierarchy is abstracted using a generic model, the Parallel Memory Hierarchy (PMH) model [2]. A programmer views a memory system as a tree, with each node representing a memory module. A Sequoia program is organized in a recursive hierarchy pattern. A program task, which operates entirely within its own private address space on a tree node, spawns child tasks onto the child nodes of the tree. Parent tasks may partition data into blocks that are to be processed by children tasks. Further, the communication between parent and children tasks through the memory hierarchy has to be explicitly expressed as arguments and results passed between parent and children tasks. The actual computations are performed by tasks on the leaf nodes.

In Sequoia, the mapping of a program hierarchy onto a memory hierarchy tree is performed by the compiler according to a mapping specification created by the programmer. A specification details how to create and map task instances for each hierarchy level in the target machine. The compiler unwinds the recursive hierarchy of a Sequoia program based on the mapping specification and creates another program that contains the static assignment of task instances to the target memory hierarchy.

The Sequoia runtime requires predefined distributions of both data and tasks at compile time. Such a requirement makes it difficult to write efficient programs for applications whose data access and computation patterns are not known until runtime. We have written graph traversal algorithms. Using common data structures, such as adjacency lists, to store a graph, we found that, without implicit

sharing of boundary elements, it is very difficult to partition the graph such that each subgraph can be processed only by a single task. Another approach is to pass the graph to each task and let each task process only a subgraph, but that approach requires excessive communication and also poses coordination issues when a task needs to notify other tasks that a given vertex was visited already.

3 Hierarchical Place Trees (HPT) Model

In the Hierarchical Place Trees (HPT) model, a memory module, such as a DRAM, cache, or device memory, is abstracted as a *place*, and a memory hierarchy is abstracted as a *place tree*. Places are annotated with attributes to indicate their memory type and size, *e.g.,* memory, cache, scratchpad, register file. A processor core is abstracted as a *worker thread*. In our current HPT model, worker threads can only be attached to leaf nodes in the place tree[1]. Figure 1 illustrates the locality-based scheduling constraints in the HPT model. As in X10, we assume that a task can be directed to place PL_i by using a statement like "*async (PL_i)*". However, unlike X10, the destination place may be an internal node or a leaf node in the hierarchy, as illustrated by the task queues associated with each place in Figure 1. If a non-leaf place PL_i is the target for an async statement in the HPT model, then the created task can be executed on any worker that belongs to the subtree rooted at PL_i. Thus, an internal node in the HPT serves as a *subtree wildcard* for the set of workers that can execute a task in its queue. For example, an "*async (PL2)*" task can be executed by worker $w2$ or $w3$. A consequence of this constraint is that a worker can only execute tasks from its ancestor places in the HPT. For example, worker $w0$ in Figure 1 can only execute tasks from the queues in places $PL3$, $PL1$, and $PL0$. If a task executing at worker $w0$ is suspended, we assume that it can be resumed at any worker (including $w0$) in the subtree of the task's original target place.
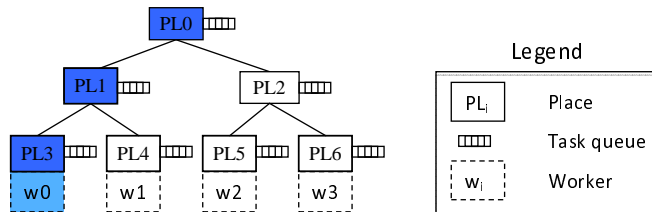

Fig. 1: Scheduling constraints in the HPT model

Figure 2 illustrates the steps involved in programming and executing an application using the HPT Model. The parallelism and locality in a program is written in a way so as to work with any *configuration specification*. (As discussed in Section 4.3, a configuration consists of an HPT model, and a mapping of the places

---

[1] In the future, we may relax this restriction and allow worker threads to be attached to internal nodes, so as to model "processor-in-memory" hardware architecture.
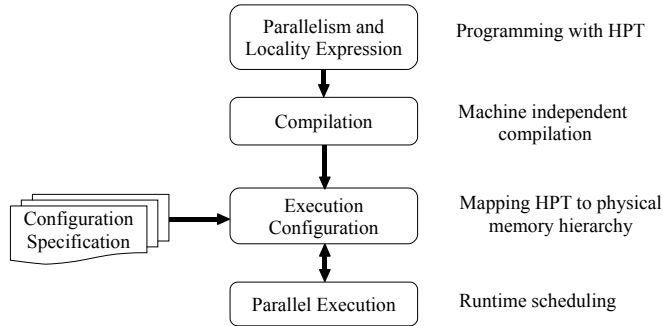
Fig. 2: Steps to program and execute an application using the HPT model

and workers in the HPT to memories and processor cores in the target machine.)
Thus, the same program can be executed with different configurations, much as
the same OpenMP or MPI program can be executed with different numbers of
processors. While it is common to use different configurations as abstractions
of different hardware systems, it is also possible to use different configurations
as alternate abstractions of the same physical machine. The best configuration
choice will depend on the application, and the desired trade-off between locality
and load balance for a given task. Auto-tuning techniques can also be used to
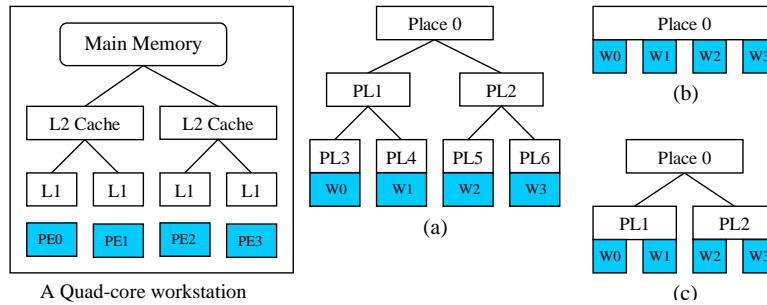help select the best configuration for a specific application and target system.

Fig. 3: A quad-core CPU machine with a three-level memory hierarchy. Figures
a, b, and c represent three different HPT configurations for this machine.

To illustrate how the HPT model can be used to obtain different abstractions
for the same physical hardware, consider a quad-core processor machine shown
in the left side of Figure 3. The hardware consists of four cores (PE0 to PE3)
and three levels of memory hierarchy. An HPT model that mirrors this structure
can be found on the right in Figure 3a. However, if a programmer prefers to view
the shared memory as being flat with uniform access, they can instead work with
the HPT model shown in Figure 3b. Or they can take an intermediate approach
by using the HPT model shown in Figure 3c.

It is challenging to develop an interface for data distribution and data transfer that is both portable and can be efficiently implemented across a range of memory systems. In SMP machines, data distribution follows implicitly from the computation mapping, whereas distributed memory machines and hybrid systems with accelerators require explicit data distributions and data transfers. To that end, the HPT model builds on the idea of a Partitioned Global Address Space (PGAS), with the extension that the partitioning is not flat and can occur across a place tree hierarchy. Further, we support three data transfer mechanisms in the HPT model: 1) data distribution with implicit data transfer; 2) explicit copyin/copyout parameters, and 3) explicit asynchronous data transfer.

3.1 Implicit Data Transfer with Data Distributions and Array Views

All data structures that are to be accessed implicitly using global addresses must have a well-defined *distribution* across places. Each scalar object is assumed to have a single *home place*. Any access to any part of the object results in a data transfer from the home place to the worker performing the access. The cost of the access will depend on the distance between the home place and the worker. Note that the programmer, compiler, runtime or hardware may choose to create a cached clone of the object closer to the worker, when legal to do so.

Distributions of arrays can be more complicated, as evidenced by the wealth of past work on this topic including array distributions in High Performance Fortran. Unlike a lot of past work on array distributions, the HPT approach to array distribution builds on the idea of *array views* [13, 12]. In this approach, a base one-dimensional array can be allocated across a set of places, and then *viewed* through a multidimensional index space. Multiple views can be created for the same base array, and may range across only a subset of the base array. A key component of an array view is the view's *distribution*, which includes the domain and range of the mapping from the view's index space to the base array. We use the [.] type notation to denote views and the [ ] type notation to denote arrays. Given an array view A, the restriction operation, $A|p$, defines a new array view restricted to elements of A contained within place p's subtree. Note that applying a restriction operator does not result in any data copying or data redistribution. Data transfer only occurs when an array view is dereferenced to access an element of the underlying array.

3.2 Explicit Synchronous Data Transfer using IN, OUT, and INOUT Parameters

A simple alternative to implicit data access is to support data transfer via explicit IN, OUT, and INOUT parameters, analogous to a dataflow model. In HPT, this is accomplished by extending the `async` construct with IN, OUT, and INOUT clauses. Upon launching a task at its destination place, the data specified by IN and INOUT clauses will be copied into the temporary space of the destination place. When the task completes, the data specified by the OUT and INOUT modifiers will be copied back to their original location. This parameter-passing approach is especially well suited to master-worker execution models on distributed-memory and hybrid systems.

3.3 Explicit Asynchronous Data Transfer

With IN, OUT and INOUT parameter semantics, the calling task blocks on the callee task until it has completed execution and all data transfers associated with the OUT clauses have completed. However, in many cases it is desirable to perform the data transfer *asynchronously* so that it can be overlapped with computation in the caller and callee tasks. To that end, we introduce an *asyncMemcpy(dest,src)* primitive in HPT, that can be used to initiate asynchronous data transfers between places. An *asyncMemcpy* call delegates the data transfer to a background worker (which could be a hardware resource such as a DMA engine) and returns with a handle of type *future⟨void⟩* that can be used to check the transfer status. As with *async* operations, there are two ways to check for termination. First, a *force()* operation can be performed to block on a specific *asyncMemcpy()* operation. Second, a *finish* statement can be used to ensure that all *asyncMemcpy()* operations launched in the scope of the *finish* have completed when execution progresses past the *finish* statement.

4 Programming Interface and Implementation

We have implemented a prototype of the HPT model as extensions of the Habanero-Java (HJ) language and runtime system [12]. HJ is a Java-based task parallel language derived from X10 v1.5 [7]. As discussed earlier, Figure 2 shows an overview of the steps involved in programming and executing an application using the HPT model. Programmers write their application using the HJ language and the HPT interfaces. Then the program is compiled using the HJ compiler and linked with the HPT and HJ runtime library. To launch the application, the runtime system requires an execution configuration (described in Section 4.3) in order to map the application computation and data access onto the hardware. During execution, the runtime system is responsible for task scheduling and data movement.

4.1 HPT Interfaces and Language Extensions

In this section, we briefly summarize the HPT interfaces and language extensions supported by our prototype implementation. Table 1 lists some of the place-based APIs available to programmers for the HPT model. We also added three clauses (IN, OUT, INOUT) to the async and ateach constructs in support of the explicit data transfer discussed earlier:

```
INOUT_Expr := [IN(Expr,...)][OUT(Expr,...)][INOUT(Expr,...)]
ASYNC_Expr := [acc] async [place_expr] [INOUT_Expr] {statements}
ATEACH_Expr := [acc] ateach [placeSet_expr] [INOUT_Expr] {statements}
```

The acc modifier is inspired by a similar keyword used for single-source programming of accelerators in [16] and related approaches. However, it has a more general meaning in the HPT model. We require that an async or ateach statement executed with the acc modifier does not perform any implicit data access outside its place; any attempt to perform such a data access will result in an

exception akin to X10's BadPlaceException [7]. While this is a requirement for execution on an accelerator like a GPGPU, the `acc` modifier can be applied to any task running on a CPU as well. All communication with an `acc` task must be performed explicitly using IN, OUT, INOUT clauses and/or *asyncMemcpy* calls.

Name	Description
dist getCartesianView(int rank)	Return a *rank*-dimensional Cartesian view of this place's child places (per-dimension factoring of children is selected by the runtime)
dist getCartesianView(int[ ] dims)	Return a Cartesian view of this place's child places using the per-dimension factors given in the *dims* array
boolean isLeafPlace ()	Return true if this place is a leaf place
Set<place> getChildren()	Return all the child places of this place
placeType getType()	Return the place's storage type (memory, cache, etc)
int getSize ( )	Return the memory size available at this place

Table 1: Subset of place-based API's in the HPT model

4.2 Programming Using the HPT Interface for Implicit Data Access

In Figure 4, we show a recursive matrix multiplication program ($C=A\times B$) written in HJ using the HPT interface. There are two portions of code in the example: the code for leaf places executed when the *isLeafPlace( )* predicate evaluates to true, and the code executed for non-leaf places otherwise.

```
1    void MatrixMult(double[.] A, double[.] B, double[.] C) {
2        if ( here.isLeafPlace( ) ) { /* compute the sub-block sequentially */
3          for (point [i,j,k] : [A.region.rank(0), B.region.rank(1), A.region.rank(1)])
4            C[i,j] += A[i,k] * B[k,j];
5        } else {
6          /* retrieve children places and structure them into a 2-D Cartesian topology, pTop */
7          dist pTop = here.getCartesianView( 2 );
8
9          /* generate array view that block-distributes C over the 2-D topology, pTop*/
10         final double[.] C_d = dist.block( C, pTop );
11         /* generate array view that block-distributes A over pTop's 1st dimension (rows) */
12         final double[.] A_d = dist.block( A, pTop, 0 );
13         /* generate array view that block-distributes B over pTop's 2nd dimension (columns) */
14         final double[.] B_d = dist.block( B, pTop, 1 );
15
16         /* recursive call with sub-matrices of A, B, C projected on to place p */
17         finish ateach( point p : pTop ) MatrixMult( A_d|p, B_d|p, C_d|p );
18        }
19    }
```

Fig. 4: Matrix multiplication example

For simplicity, this example only uses implicit data accesses through array views. The views, *A_d*, *B_d* and *C_d*, are used to establish the subregions for recursive calls to *MatrixMult()* via *restriction* operators of the form *A_d|p*. Note that creating views does not result in a redistribution of the arrays. Instead, the use of the *ateach* construct in line 17 has the effect of establishing an affinity (akin to tiling) among iterations through the recursive structure of *MatrixMult()*.

4.3 Execution Configurations

As shown in Figure 2, an HJ program written using the HPT interface can be executed on different systems, and with different *execution configurations*. The configuration specification is supplied as an XML file, and describes the target machine architecture as a physical place tree (PPT) as well as a mapping of the HPT to the PPT. Figure 5 shows the PPT specification for the quad-core workstation shown in Figure 3. In our approach, the mapping is performed when launching the program. This is different from the Sequoia approach in which the mapping is performed by the compiler, thereby requiring a recompilation to generate code for each new hardware configuration.

```
1    <ppt:Place id="0" type="memory" xmlns:ppt="http://habanero.rice.edu/pptl" ... >
2        <ppt:Place id="1" type="cache" size="6291456" unitSize="128"> <!-- L2 cache -->
3            <ppt:Place id="3" type="cache" cpuid="0">
4                <ppt:Worker id="0" cpuid="0"/> </ppt:Place>
5            <ppt:Place id="4" type="cache" cpuid="1">
6                <ppt:Worker id="1" cpuid="1"/> </ppt:Place> </ppt:Place>
7        <ppt:Place id="2" type="cache" size="6291456" unitSize="128"> <!-- L2 cache -->
8            <ppt:Place id="5" type="cache" cpuid="2">
9                <ppt:Worker id="2" cpuid="2"/> </ppt:Place>
10           <ppt:Place id="6" type="cache" cpuid="3">
11               <ppt:Worker id="3" cpuid="3"/> </ppt:Place> </ppt:Place>
12   </ppt:Place>
```

Fig. 5: Physical place tree specification for a quad-core workstation

In Figure 5, the *type* attribute is used to specify the type (memory, cache, or accelerator) of the memory module the place represents. The *size* attribute specifies the place's storage size (cache or memory). The *cpuid* attribute is only valid for a worker and is used as a target for mapping HPT worker threads.

4.4 Unified CPU/GPU Code

As mentioned earlier, the acc modifier for async and ateach asserts that no implicit data access will be performed outside the task's local place. Such activities are suitable for execution on hardware accelerators, such as GPUs, if available, but can also run on CPUs if so desired. However, the converse is not true i.e., a task without an acc modifier cannot be executed on a GPU. For GPU executions, our prototype implementation leverages the JCUDA compiler and runtime library [11] for generating Java glue code and JNI stub functions that handle data transfer between CPU memory space and GPU memory space. The current HPT implementation restricts the use of acc keyword to occur with nested ateach and foreach statements that facilitate code generation for a GPU.

Figure 6 shows the usage of the acc keyword with the INOUT clause in the CPU/GPU unified code for the Java Grande Forum Series benchmark. Figure 7 shows the compiler transformed pseudo-code after the first stage of analysis, mainly converting the nested ateach and foreach to an *if-else* branch. The *top.isCUDAGrid( )* condition of the branch evaluates to true if the current topology is a CUDA grid configuration, thereby splitting the code into two paths: the GPU path and the CPU path.

```
1    double[] baseArray = new double[ [0:1, 0: array_rows-1] ];
2    double[.] testArray = dist.blockView(baseArray);
3
4    void doSeries( ) {
5        dist top = here.getCartesianView(1);
6        finish acc ateach( place p : top ) INOUT (testArray) {
7            foreach( point i : testArray | p ) seriesKernel( testArray, i ); }
8    }
```

Fig. 6: Unified CPU/GPU code for Series example using INOUT

```
1    void doSeries( ) {
2      dist top = here.getCartesianView(1);
3
4      if ( top.isCUDAGrid( ) ) {
5        int [ ] dimBlocks = { 256, 1, 1 }; // 256 is the desired block size
6        int [ ] dimGrids = { ( array_rows + dimBlocks[0] - 1 ) / dimBlocks[0], 1, 1 };
7        /* The definition of seriesKernel (not shown) specifies an INOUT attribute for testArray */
8        seriesKernel_stub.seriesKernel <<<< dimGrids, dimBlocks >>>>  ( testArray, array_rows );
9      } else {
10        final dist d = dist.block( testArray, top );
11        finish ateach( place p : top )  {
12          foreach( point i : [d|p.rank(0).low( ) : d|p.rank(0).high( )] )
13                  seriesKernel( testArray, i ); }
14     }
15   }
```

Fig. 7: Compiler-generated pseudo-code for Series benchmark from Figure 6 using JCUDA for the GPU path

5   Preliminary Experimental Results

In this section, we evaluate a prototype implementation of the HPT model in the HJ system with the JCUDA extension for GPU's [11]. The experimental results were obtained on three system architectures: a Niagara T2, a Xeon SMP, and a hybrid system with NVIDIA Tesla C1060 GPGPUs. The Niagara T2 has a 8-core UltraSPARC T2 processor clocked at 1.2GHz, and 32GB main memory. Each core supports 8 hardware threads, and all cores share a single 4MB L2 cache with 8 banks. The Xeon SMP has four Quad-Core Intel Xeon E7330 processors running at 2.40GHz with 32GB main memory. Each quad-core processor has two core-pairs and each core-pair shares a 3MB L2 cache. A single NVIDIA Tesla C1060 GPGPU has 4GB memory and 240 cores in 30 streaming multiprocessors (SM's), running at 1.3 GHz clock speed. Each SM has 8 scalar cores and 1 double-precision FPU, and 16KB shared (local) memory.

The benchmarks used for evaluation include the 2D Successive Over-Relaxation (SOR) and IDEA encryption (Crypt) benchmarks from the Java Grande Forum Benchmark suite [6], and Conjugate Gradient method (CG) from NAS Parallel Benchmarks [3]. Both CG and SOR2D are locality-sensitive applications as their computation kernels include matrix and vector operations that exhibit both spatial and temporal reuse. For GPU evaluation, we also include a Matrix Multiplication (MatrixMult) kernel that computes the product of two $1680{\times}1680$ dense single-precision matrices. The Crypt benchmark is an example of a streaming ap-

plication, and was chosen as a counter-point to SOR, CG, and MatrixMult. We added one more size, D (100M), to study the impact on large data size streaming. For evaluating HPT on cache memory hierarchy, we use data distribution for data sharing; and for evaluation on GPUs, we use IN, OUT and INOUT to specify data transfer between places.

The JVM used for evaluation is Sun JDK 1.6, invoked with the following options: -Xmx2g -Xms2g -server -Xss8M -XX:+UseParallelGC -XX:+AggressiveOpts. For all results shown below, we report the average performance across 20 executions so as to reduce the impact of JIT compilation and garbage collection.

5.1 Evaluation on CPU Memory and Cache Architecture

To evaluate the HPT model for CPU memory and cache architecture, we created three execution configurations for each of the Niagara T2 and Xeon SMP. The configuration details are listed in Table 2.

Machine	Config	Description	Modeling
Niagara T2	1x64	One root place with 64 leaf places	Each leaf place represents the L1 cache of a T2 SMT thread
	8x8	8 non-leaf places, each of which has 8 leaf places	The 8 non-leaf places represent 8 L2 cache banks; the 8 leaf places of each non-leaf place represent the 8 SMT thread
	64x1	64 non-leaf places, each of which has 1 leaf place	The 64 non-leaf places represent the L1 cache of the total 64 SMT threads
Xeon SMP	1x16	One root place with 16 leaf places	Each leaf place represents the L1 cache of each of the 16 cores
	8x2	8 non-leaf places, each of which has 2 leaf places	The 8 non-leaf places represent the 8 L2 caches of the 8 core pairs; the 2 leaf places of each non-leaf place represent the 2 L1 caches in each core pair
	16x1	16 non-leaf places, each of which has 1 leaf place	The 16 non-leaf places represent the L1 caches of the total 16 cores

Table 2: Execution configurations for Niagara T2 and Xeon SMP

The 8x2 configuration for the Xeon SMP and the 8x8 configuration for the Niagara T2 most closely model sharing at the L2 cache level for these two machines. Table 3 includes the execution times in seconds for three benchmarks with different problem sizes using three configurations on the two machines. On the Xeon SMP, the 8x2 configuration performs much better than the other configurations for sizes A, B, C of SOR2D and size A of CG. For CG size B, the performance was similar for all three configurations suggesting that the benefits of sharing may decrease as the data size increases. As expected, there was no significant difference in performance across the three configurations for the Crypt benchmark. In fact, there was a minor performance degradation for the 8x2 configuration relative to the others for Crypt size D.

The results for Niagara T2 showed some interesting variations. As may be expected, the 8x8 configuration performs better than the other two configurations

for SOR2D Size C, CG Size A and B, and Crypt Size C and D. However, unlike for the Xeon SMP, the 8x8 configuration exhibits much better locality relative to the other two for CG size B. We are still in the process of analyzing the reason for this improvement. However, we observe that the L2 cache structure is very different in the Xeon and Niagara T2, and it's possible that bank effects may contribute to the improvement obtained by the 8x8 configuration.

Benchmark	Size	Xeon SMP			Niagara T2		
		1x16	8x2	16x1	1x64	8x8	64x1
SOR2D	A	0.42	0.20	0.84	1.22	1.83	4.10
	B	0.76	0.28	1.74	2.12	2.63	7.67
	C	1.14	0.42	1.90	3.77	3.49	10.85
CG	A	1.34	0.61	0.70	4.79	3.36	6.01
	B	39.35	38.46	39.41	110.22	51.78	123.55
Crypt	B	0.41	0.41	0.42	1.95	2.29	2.34
	C	0.91	0.92	0.90	4.23	4.11	4.28
	D	3.05	3.17	3.05	5.51	5.06	5.79

Table 3: Execution times in seconds for different configurations

In general, the performance impact of using different execution configurations are more significant for locality-sensitive applications (SOR 2D and CG) than that for locality-neutral applications (Crypt). Among the benchmarks that perform better in Xeon SMP when using the 8x2 configurations, the SOR 2D Size A and B benchmarks perform worse in Niagara T2 when using the 8x8 configurations. In the Niagara T2 8x8 configuration, the 8 leaf places of each non-leaf place share the core L1 cache (8KB); while in the Xeon SMP 8x2 configuration, each of the 2 leaf places of a non-leaf place has its own L1 cache (128KB). This may be one factor that induces the speedup differences.

5.2 GPU Evaluation

We evaluate the HPT model with JCUDA extensions on GPUs using the Matrix-Mult kernel and SOR 2D benchmarks. We varied the number of threads per block and blocks per grid in CUDA kernel invocations for each data set of the benchmarks. In each execution, the total number of threads for those configurations are the same. We obtain the execution time of each configuration for comparison, as plotted in Figure 8.

In Figure 8a, the best performance obtained for multiplying two 1680x1680 matrices is using the configuration of 16x16 threads/block and 105x105 blocks/grid. We also noticed significant execution time differences of using other configurations. In the results for SOR 2D benchmark, shown in Figure 8b, the execution times of using different configurations vary slightly as compared to the matrix multiplication example. Yet they still follow the same pattern and the fastest occurs when using the 120x64 configurations. We conclude that choosing the correct configuration in GPUs is crucial in achieving good performance.

In Figure 9, we show the results of varying blocks/grid for SOR and Crypt to study the performance variation relative to the geometry of blocks/grid. We have
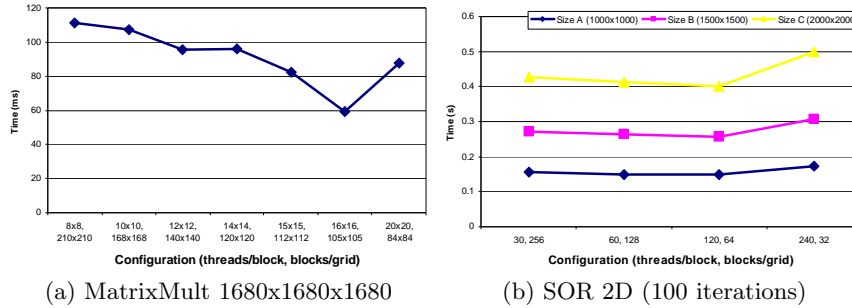
(a) MatrixMult 1680x1680x1680

(b) SOR 2D (100 iterations)

Fig. 8: Execution time by varying threads/block and blocks/grid on Tesla C1060

(a) SOR (256 threads/block)
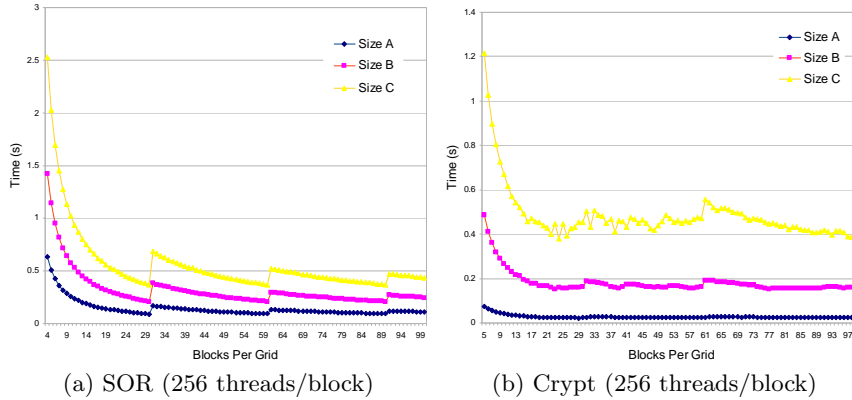
(b) Crypt (256 threads/block)

Fig. 9: Execution time by varying blocks/grid (256 threads/block) on Tesla C1060

chosen 256 threads/block, a number that is commonly used to enable GPUs to maintain high occupancy of its cores [15]. We observe the lowest execution times when blocks/grid is a multiple of 30, and the degradation when the blocks/grid is 1 more than a multiple of 30 (31, 61, 91, . . .). As the Tesla C1060 has 30 streaming multiprocessors (SM), a multiple of 30 is the configuration that enables high thread throughput as all cores of a SM are occupied in each scheduling cycle.

6 Related Work

Modern parallel language efforts (e.g. X10 [7], Chapel [1], and Fortress [14]) provide language-based approaches, instead of library extensions, to specifying concurrency and data distributions. As described in the Background section, the X10 "place" and Chapel "locale" concepts provide an abstract model for co-locating tasks and data; however, this model is flat in structure and cannot capture vertical locality in a memory hierarchy. For Sequoia's approach [5], a computation task and the task's processing data are combined to be assigned

to the tree leaf nodes at the compilation time. Our HPT model operates these at the runtime as it first performs data distribution among multiple places, and then schedules tasks on places according to the affinity between tasks and place resident data. In Sequoia, accesses to data, which are passed as task parameters, have to be localized and explicit. In HPT model, we also allow for a task to implicitly access data that resides at upper-level places in the hierarchy tree.

Bikshandi et al [4] proposed Hierarchically Tiled Array (HTA) to facilitate the direct manipulation of tiles across processors. By permitting arbitrary element access to a distributed array, the HTA model concentrates on tiling the array data and exports this explicit information to the compiler to partition loops for locality or parallelism. The HPT model is more general since the hierarchical place tree can be used for both data and computation, and can support distribution of data structures that go beyond arrays.

The Concurrent Object Oriented Language (COOL) [8] is an example of a programming model with locality annotations that can be used by the compiler and runtime. It extends C++ with concurrency and data locality abstractions for programmers to supply hints about the data objects referenced by parallel tasks. These hints are used by the runtime system to appropriately schedule tasks and migrate data, and thereby exploit locality in the memory hierarchy.

Compared with past work, the HPT model leverages advantages from both Sequoia and X10. Programmers have the flexibility to express the concurrency and locality in their application using a portable hierarchical place tree structure. Deployments are a first-class construct in the HPT approach, and provide the key to bridging from a single application to different parallel hardware platforms.

7 Conclusions and Future Work

The Hierarchical Place Tree (HPT) model provides a generalized abstraction of memory hierarchies and parallel system. Starting as a fusion of concepts from the Sequoia and X10 programming models, the HPT approach results in capabilities that are not supported by either. We demonstrated the use of the HPT model as a vehicle for portable programming of unified CPU/GPU codes. The evaluation results show performance improvement for locality-sensitive applications when they are executed with configurations that are consistent with the system memory architectures.

There are several directions for future work, and we briefly mention some below. First, we are investigating the use of profiling tools that can isolate memory and cache accesses to better understand the locality impact of the HPT approach. Second, we believe that it is important to develop a theoretical model for quantifying data locality in the HPT model as was done in the PMH model [2, 10]. Third, it is important to investigate the feasibility of the HPT model for cluster environments in which data movement and distribution are more constrained and have a more significant impact on performance than that in shared-memory systems. Finally, we would like to explore new compiler analysis and optimization techniques to improve locality and parallelism in the HPT context.

Acknowledgments

References

1. Chapel Programming Language. http://chapel.cray.com/.
2. B. Alpern et al. Modeling parallel computers as memory hierarchies. In *In Proc. Programming Models for Massively Parallel Computers*, pages 116–123. IEEE Computer Society Press, 1993.
3. D. H. Bailey et al. Nas parallel benchmark results. In *Proceedings of SC'92*, pages 386–393, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.
4. G. Bikshandi et al. Programming for parallelism and locality with hierarchically tiled arrays. In *Proceedings of PPoPP'06*, pages 48–57, New York, NY, USA, 2006.
5. K. Fatahalian et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the SC'06*, 2006.
6. L. A. Smith et al. A Parallel Java Grande Benchmark Suite. In *SC'01*, pages 8–8, New York, NY, USA, 2001. ACM.
7. P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA 2005 Onward! Track*, 2005.
8. R. Chandra et al. Cool: An object-based language for parallel programming. *Computer*, 27(8):13–26, 1994.
9. S. Chandra et al. Type inference for locality analysis of distributed data structures. In *PPoPP '08*, pages 11–22, New York, NY, USA, 2008. ACM.
10. Xin-Min Tian et al. Quantive studies of data-locality sensitivity on the EARTH multithreaded architecture: preliminary results. *International Conference on High-Performance Computing*, 1996.
11. Y. Yan et al. JCUDA: a Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Proceedings of Euro-Par 2009*, Delft, The Netherlands, August 2009.
12. Habanero Java (HJ) Project. http://habanero.rice.edu/hj.
13. Mackale Joyner. *Array Optimizations for High Productivity Programming Languages*. PhD thesis, Houston, TX, USA, 2008.
14. Sun Microsystems. Fortress Programming Language. http://projectfortress.sun.com.
15. NVIDIA. *CUDA Programming Guide 2.2*. 2009. http://www.nvidia.com/cuda.
16. The Portland Group. PGI Fortran & C Accelerator Compilers and Programming Model. Technical report, November 2008.