Abstract

General purpose computing architectures are evolving quickly to become many-core and hierarchical: i.e. a core can communicate more quickly locally than globally. To be effective on such architectures programming models must be aware of the communication hierarchy, and yet preserve performance portability.

We propose four new architecture-aware constructs for the parallel Haskell extension GrH that exploit information about task size and aim to reduce communication for small tasks, preserve data locality, or to distribute large units of work. We report a preliminary investigation of architecture-aware programming models that abstract over the new constructs. In particular we propose architecture-aware evaluation strategies and skeletons. We investigate three common parallel paradigms on hierarchical architectures with up to 224 cores. The results show that the architecture-aware constructs consistently deliver better speedup and scalability than existing constructs together with dramatically reduced variability. In some experiments speedup is improved by an order of magnitude.

Keywords: Architecture-aware, Runtime, System

1. Introduction

General purpose computing architectures are inexorably exploiting many-cores, with the number of cores following Moore’s Law. Future architectures will inevitably have a hierarchical, or tree-like, communications structure. To exploit such architectures, programming models must be aware of the communication hierarchy. Given the rate of architecture evolution it is important that the programming model preserves performance portability as far as possible.

Glasgow Parallel Haskell (GrH) [1, 2] is a well-established, and widely used parallel extension to Haskell. Parallelism in the GrH model is achieved using only two constructs, par and paraq. Evaluation strategies are higher-order, polymorphic functions that provide high-level control of parallelism by abstracting over the constructs [3]. Currently the GrH programming model is oblivious to the underlying architecture: any unit of work may be executed by any core.

The paper makes the following research contributions. (1) We propose four new architecture-aware constructs for GrH on architectures with hierarchical communication. The constructs aim to control data locality and work
distribution guided by information about task size. They preserve performance portability by exposing a virtual, rather than physical, architecture, and minimising prescription e.g. allowing the implementation to place a task on any one of a set of cores (Section 3). (2) We investigate whether the new architecture-aware constructs can deliver improved performance on hierarchical architectures, considering simple examples of common paradigms: data parallel, divide-and-conquer, and nested parallelism. The evaluation is based on architectures up to 224 cores (Section 5). (3) We make a preliminary investigation into architecture-aware programming models that abstract over the new constructs. In particular we propose architecture-aware evaluation strategies, and architecture-aware skeletons. The abstractions aid performance portability by isolating architecture-specific aspects of the program. Some of the new abstractions are used in the programs measured, and initial performance results are promising (Section 4).

2. Background

2.1. The Trend Towards Hierarchical Architectures

Nowadays, the most common parallel architectures are clusters of multicore nodes, with three levels in the hierarchy: threads on the same core can communicate much quicker with a thread on the same core, more slower with a thread on another core in the node, and even slower still with threads on remote nodes.

Figure 1 illustrates both a virtual and a real hierarchical architecture. The virtual architecture comprises a tree, possibly unbalanced, and where the degree of the nodes may vary. We expect communication at lower levels in the tree to be faster than at higher levels.

2.2. Glasgow Parallel Haskell (GpH)

Primitives. Pure parallelism in Haskell is achieved using only two constructs, \texttt{par} and \texttt{pseq}, both with type \texttt{a \rightarrow b \rightarrow b}. The \texttt{par} combinator introduces a potential for parallel evaluation. When \texttt{par} is applied to two arguments, it returns the value of its second argument, while its first argument is sparked to possibly be evaluated in parallel. Operationally the computation is placed in the processing element’s (PE’s) spark pool.

In general, it is not enough to provide \texttt{par} alone, because when suggesting that something is to be evaluated in parallel, it is useful to be able to say what it is to be evaluated in parallel with. Haskell neither specifies nor requires a particular order of evaluation, so normally the programmer has no control over this aspect of their program’s execution. In particular, the programmer has no control over when a particular call to \texttt{par} will be evaluated, or what will be evaluated before or after it (or indeed in parallel with it). This is the reason for \texttt{pseq}: a call \texttt{pseq a b} introduces an order-of-evaluation requirement that \texttt{a} be evaluated before \texttt{b}.
data Eval a = Done a

instance Monad Eval where
  return x = Done x
  Done x >>= k = k x

runEval :: Eval a -> a
runEval (Done a) = a

type Strategy a = a -> Eval a

using :: a -> Strategy a -> a
x 'using' s = runEval (s x)

dot :: Strategy a ->
      Strategy a ->
      Strategy a
s2 'dot' s1 = s2 . runEval . s1

rseq :: Strategy a
rseq x = x 'pseq' return x

rpar :: Strategy a
rpar x = x 'par' return x

evalList :: Strategy a -> Strategy [a]
evalList s [] = return []
evalList s (x:xs) = do
  x' <- s x;
  xs' <- evalList s xs;
  return (x':xs')

parList :: Strategy a -> Strategy [a]
parList s = evalList (rpar 'dot' s)

parMap :: Strategy b->(a -> b)->[a]->[b]
parMap s f xs = map f xs 'using' parList s

Figure 2: The Essence of the Evaluation Strategies

Evaluation Strategies. Evaluation strategies [1], or “strategies” for short, are a key abstraction for adding pure, deterministic, parallelism to Haskell programs. Figure 2 shows the Strategies that use an Eval monad, and a Strategy on some type a is a function to Eval a. The using function applies a strategy, and dot composes strategies. rpar and rseq are strategies corresponding to par and pseq respectively.

Parallel specifications can be built up in a compositional way, for example evalList s sequentially applies strategy s to every element of a list. Similarly parList s applies the strategy s to every element of the list in parallel. Moreover, the parallelism can be specified independently of the main computation, so for example the computational part of parMap is unchanged and isolated from the parallel control, a parList.

Load Management in GpH/GUM. If a PE becomes idle, it will extract a spark from its local spark pool if it can, and use this to create a new thread. If there are no useful local sparks, then the PE sends a FISH message that is passed at random from PE to PE until some work is found. If the PE that receives a FISH has a useful spark it sends a SCHEDULE message to the PE that originated the FISH, containing the corresponding graph.

Problems of GpH. The current load management along with parallel recognitions mechanism in GpH does not provide a way of controlling the data locality and task placement that deals with the new hierarchical architectures. The implication of this issue is the execution of small tasks far from the creator, which does not cover the overhead. This has real impact on performance. Even the solution provided by strategies which tend to group small tasks to form large tasks, it is not suitable for hierarchical architectures. Therefore, only if the underlying hierarchical architectures is respected by the code, can efficient execution be expected [4]. In the following, we will describe our approach for programming hierarchical architectures.

3. New Architecture-Aware Constructs

3.1. Virtual Architectures

For some problems, like matrix manipulations, optimal performance can be obtained on a specific architecture by explicitly placing threads within the architecture. However, many problems do not exhibit this regularity. Moreover, explicit placement prevents performance portability: the program must be rewritten for a new architecture, a crucial deficiency in the presence of fast-evolving architectures.

To avoid these deficiencies we, like others [5], propose language constructs that expose a virtual architecture rather than the actual architecture. Clearly the virtual architecture must be readily mapped to physical architectures.
parDist :: Int -> Int -> a -> b -> b
parBound :: Int -> a -> b -> b
parAtLeast :: Int -> a -> b -> b
parExact :: Int -> a -> b -> b

parBound n x y = parDist 0 n x y
parAtLeast n x y = parDist n maxLevel x y
parExact n x y = parDist n n x y

Figure 3: Architecture Aware Constructs (on left) and Definition (on right)

In addition, our constructs minimise prescription: they identify sets of locations where the thread may be placed. Moreover, we support performance portability by isolating the architecture-specific parts of the program in just a few functions that can be refactored for a new architecture.

3.2. Mapping Parallelism to an Architecture

Broadly speaking there are two challenges to be solved simultaneously when controlling parallelism on a hierarchical architecture:

Limit the communication costs for small computations. This entails limiting how far small computations are communicated, and requires information about thread granularity (i.e. execution time). Without this information, programs often have poor resource utilisation as the system is saturated with small threads. Thread granularity information may be obtained from a number of sources, for example, from some resource analysis, by profiling, by the programmer. We do not address the problem of obtaining this information here. Although the examples in the following section use granularity information from program parameters, we have adapted the runtime system to store granularity information with each spark if required.

Keep all cores busy, for example, at system start up we must quickly distribute work to all cores. This entails sending large-grain computations long distances over the communications hierarchy.

3.3. New Constructs

The new architecture-aware constructs are summarised in Figure 3. The constructs identify sets of locations where a computation may be performed, and the runtime system is free to place the computation within this set. These sets often include multiple levels in the communication hierarchy. All of the constructs are implemented using the parDist primitive in the GUM runtime system that creates sparks labelled with minimum and maximum communication distances.

To limit the communication costs for small computations, or to preserve data locality, we propose parBound that behaves like par, except that it takes an additional integer parameter specifying the maximum distance in the communication hierarchy that the computation may be located. The distance represents a level in the hierarchy illustrated in Figure 1. parBound illustrates a key characteristic of our constructs, namely that while placing restrictions on how work is communicated, they aim for minimal prescription.

For example, parBound 0 means the computation may not leave the core, parBound 1 may be located within the shared memory node, parBound 2 that may be located to another node in the Beowulf cluster, and parBound 3 that it may be located freely to any core appear in level 3 or below of the machine. More specific, a computation sparked by a parBound 1 may be placed on any core in the shared memory node, and a computation sparked by a parBound 2 may be placed on any core in the set identified in Figure 1.

A parAtLeast is a dual to parBound, as it takes an additional integer parameter specifying the minimum distance in the communication hierarchy that the computation may be communicated. The idea is to communicate large-grain computations large distances.

parDist can also be used to define other constructs. By way of illustration parExact in Figure 3 specifies an exact level, although not a specific processor. A parDist can also be parameterised to capture other notions, for example parDist (n-1) n specifies that a spark may be communicated to a core residing at either level (n-1) or level n, and parDist (n-2) n is similar. This provides more flexibility as there is a bigger set of locations that can fish the spark away.
4. Architecture Aware Programming

The functional programming philosophy is to construct high-level, often higher-order, programming constructs by abstracting over the constructs. We have started to investigate both architecture-aware skeletons and evaluation strategies.

```haskell
parFibDist :: Int -> Int -> Int
parFibDist t n
  | n < t = nfib n
  | otherwise = res
where
  x = parFibDist t (n-1)
  y = parFibDist t (n-2)
  res = parDist mn mx x (y 'pseq' (x+y+1))
  (mn,mx) = findLevel n

findLevel :: a -> (a,a)
findLevel x
  | (x <= 46) =(0,0)
  | (x == 47) =(1,1)
  | (x == 48) =(2,2)
  | otherwise =(3,3)
```

Figure 4: parFibDist program (on left) and findLevel program (on right)

4.1. Determining thresholds

To obtain reasonable performance, a key issue is to determine the threshold thread granularity values in the findLevel function. One approach, and that used for these benchmarks, is to determine the threshold values by experimentation. The threshold value can be based on the input value or the depth of recursion. Figure 4 shows a definition of the findLevel function based on input value and how can be use in Fibonacci function. In terms of parFibDist performance we accept the Allparam results presented in Section 5.3, because it uses similar function as source of divide-and-conquer parallelism. Alternatively, an automatic runtime system level solution is possible if the thread granularity of each spark can be identified, for example by program or resource analysis. Then suitable thread granularity thresholds can be determined for each level in a given architecture, for example using a benchmarking suite.

```haskell
rparDist :: Int -> Int -> Strategy a
rparDist min max x = parDist min max x (return x)
parDistList :: Int -> Int -> Strategy a
  -> Strategy [a]
parDistList mn mx strat (x:xs) = do
  let
    x' <- rparDist mn mx (x 'using' strat )
    xs' <- parDistList mn mx strat xs
    return (x':xs')
```

4.2. Architecture Aware Strategies

Section 5 evaluates the constructs and identifies parDist and parExact as the most suitable. To facilitate programming we must develop higher level abstractions like evaluation strategies. Defining a strategic version of the new constructs is straightforward as shown in Figure 5. Both parDistList and parListLevel are analogous to parList. However, care must be taken when refactoring the strategies module both to preserve the existing strategies, and to make the higher-order strategies parametric in the new constructs. For example, preserving a data locality with parDistList is a straightforward, we simply call the parDistList strategy with 0 and 1. The idea is to evaluate the list of computations on the same multicore machine. rparDist is used to define skeletons in Figures 6 and 7. parListLevel is used in the sumEuler and parDistList is used in the Queens program measured in Section 5.6 and listed in [6].
parMapLevel fl f (x:xs) = do
  let
    (mn,mx) = fl x
    b <- rparDist mn mx (f x)
    bs <- parMapLevel fl f xs
  return(b:bs)

Figure 6: A parMapLevel Skeleton

divCon trivial solve divide combine findlevel depth arg
  | trivial arg depth = solve x
  | otherwise = runEval $ (do
    let
      (left,right) = (divide arg)
      mx = findlevel depth
      x' <- (rparDist mx mx 'dot' rdeepseq)
        (divCon trivial solve divide combine findlevel (depth-1) right)
      y' <- (rparDist mx mx 'dot' rdeepseq)
        (divCon trivial solve divide combine findlevel (depth-1) left)
    return(combine x' y')

Figure 7: A Divide-and-conquer Skeleton

4.3. Architecture Aware Skeletons

Algorithmic skeletons capture common patterns of parallelism and are widely used, e.g. [7, 8]. We have adapted some skeletons to use architecture-aware constructs and strategies, specifically parallel map and divide-and-conquer skeletons.

An Architecture Aware Parallel Map Skeleton. The common parallel map is adapted as in Figure 6, it takes an additional findLevel parameter and applies rparDist rather than rpar. parMapLevel is used in all of the programs measured in Section 5 including the Allparam program listed in Figure 8.

A Divide-and-conquer Skeleton. We give the implementation of architecture-aware constructs in a divide-and-conquer pattern as in Figure 7. The skeleton is parameterised with the typical control functions for divide and conquer which decide if a subproblem is trivial, how to solve a trivial subproblem, how to divide a non-trivial problem and how to combine results of subproblems. Additionally, we provide a findLevel function along with depth. findLevel is a new parameter to exploit the communication architecture. The depth is to limit the amount of parallelism, by evaluating children below the depth sequentially. The performance of the skeletons is evaluated in Section 5.6.

5. Architecture Aware Construct Evaluation

Our measurements are made on an architecture comprising a Beowulf cluster of 8-core multicore nodes with a freestanding 8-core multicore (lxpara3). The communication hierarchy has four levels, so from a Beowulf core Level 0 is on the same core; level 1 is to another core on the same node; level 2 is to a core on another Beowulf node; level 3 is to a core on the freestanding multicore (lxpara3). Machine lxpara3 is an eight-core 8GB RAM, workstation comprising two Intel 5410 processors each running at 2.33GHz. The 32 Beowulf cluster nodes each comprise eight Intel 5506 cores running at 2.13GHz. All machines running under Linux CentOS 5.5. The Beowulf nodes are connected with Baystack 5510-48T switch with 48 10/100/1000 ports. Both Beowulf and lxpara3 are connected to the network with Extreme Networks Summit 400-48t, 48 10/100/1000BASE-T, 4 mini-GBIC, Extremeware.

We first investigate whether the new architecture-aware constructs can deliver improved performance on a hierarchical architectures. We do so by considering common paradigms namely data parallel, divide-and-conquer, and nested parallelism.
5.1. Divide-and-Conquer Parallelism

To investigate the new architecture-aware constructs for divide-and-conquer parallelism, we use the parFibDist version of the parallel nfib function shown in Figure 4. Here \( n \) is the Fibonacci number \( t \) is the threshold value below which we use sequential computation (nfib). Above the threshold value, \( x \) is sparked with a parDist parameterised by levels computed by findLevel.

```
mapparfib n t xs = parMapLevel findLevel (parFibOneLevel t) randomList
  where
    randomList = take n (filter (>35) xs)

parFibOneLevel ::Int-> Int -> Int
parFibOneLevel t n
  | n < t = nfib n
  | otherwise = res
  where
    x = parFibOneLevel t (n-1)
    y = parFibOneLevel t (n-2)
    res= parBound 1 x (y `pseq` (x+y+1))
```

Figure 8: Allparam Program

5.2. Data Parallelism

To investigate the new architecture-aware constructs for data parallelism, we use two programs. The intention for both programs is to generate data parallel tasks of random thread granularity. Both programs compute some function on every element of a list. Code for key functions of both programs is given in [6]. The first program, parMapList splits the list into sublists of random sizes, and the second program, parMapIntervals splits the interval into subintervals of random sizes, and the variation in task sizes, between 0.14s and 40.47s for parMapList, between 0.19s and 55.00s for parMapIntervals, and between 0.14s and 2.90s for Allparam. The difference between the programs is that parMapList communicates the list, where the parMapIntervals communicates only the start and end points of the interval. Both programs compute the sum of the Euler totient function on each list interval.

5.3. Nested Parallelism

To investigate the new architecture-aware constructs for nested parallelism, we use the Allparam program in Figure 8 with top level data parallelism and nested divide-and-conquer parallelism. More specifically it maps the parallel divide-and-conquer Fibonacci function parFibOneLevel over a list of random integers in parallel using parMapLevel. The parMapLevel function is used to distribute the work among machines while the parFibOneLevel is used to localise its work in a same multicore machine.

5.4. Results

Both data parallel programs have similar performance. Figure 9 shows the absolute speedups of parMapIntervals program measured on 8, 16, 32, 64,..., 224 cores. We make the following observations:

(1) All of the architecture-aware constructs scale: as the number of processors increase, the speedup increases. The architecture-aware constructs consistently perform better than par: by a factor between 2.5 and 10.2 on 224 cores. (2) The par program has worse performance because it has significantly higher communication overheads than the other constructs programs e.g. on 224 cores. The par exchanges 13360 messages where the parExact programs exchanges just 2292 messages. (3) While parExact gives good performance, it is vulnerable to adverse scheduling because it identifies a specific level. Less prescriptive constructs like parDist (n-2) \( n \) can give better performance on large architectures, e.g. parMapIntervals on 32 and 64 cores. (4) All of the architecture-aware constructs scale, but par however does not scale beyond 32 cores. The reason is par does not provide any restriction on thread placement and hence small thread can be executed remotely which does not cover the communication cost. (5) In both graphs parExact scales more smoothly, but parDist (n-1) \( n \) ultimately delivers the best performance.
(6) In absolute terms parMapIntervals has 20% better performance than parMapList as it communicates only a pair of numbers, rather than a list segment.

Table 1: Speedups

<table>
<thead>
<tr>
<th>NoPe</th>
<th>parMapList</th>
<th>parMapIntervals</th>
<th>Allparam</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>parExact</td>
<td>parDist (n-1) n</td>
<td>parExact</td>
</tr>
<tr>
<td>par</td>
<td>Exact (n-1) (n-2)</td>
<td>par</td>
<td>Exact (n-1) (n-2)</td>
</tr>
<tr>
<td>8</td>
<td>6.1</td>
<td>5.9</td>
<td>5.8</td>
</tr>
<tr>
<td>16</td>
<td>12.1</td>
<td>11.7</td>
<td>11.9</td>
</tr>
<tr>
<td>32</td>
<td>13.8</td>
<td>22.0</td>
<td>22.9</td>
</tr>
<tr>
<td>64</td>
<td>11.8</td>
<td>38.3</td>
<td>34.7</td>
</tr>
<tr>
<td>96</td>
<td>10.4</td>
<td>42.9</td>
<td>42.1</td>
</tr>
<tr>
<td>128</td>
<td>8.9</td>
<td>48.8</td>
<td>38.4</td>
</tr>
<tr>
<td>192</td>
<td>5.9</td>
<td>47.1</td>
<td>37.0</td>
</tr>
<tr>
<td>224</td>
<td>5.0</td>
<td>37.4</td>
<td>50.4</td>
</tr>
</tbody>
</table>

Table 1 summaries the performance of the new constructs in comparison with par. The three programs are measured on 8, 16, 32, 64, ..., 224 cores. The absolute speedups of parMapList and parMapIntervals are calculated with respect to the optimised sequential runtime (2252.60s) for the larger problem size of 140000. Data for the other constructs (parBound and parAtLeast) and smaller numbers of cores variability, runtime curves, and speedups curves can be found in [6]. The main observation is par has poorer performance because of that par program has significantly higher communication overhead than the programs using architecture-aware constructs. E.g. on 224 the par exchanges 13360 messages where the parExact program exchanges just 2292 messages. While parExact gives good performance, it is vulnerable to adverse scheduling because it identifies a specific level. Less prescriptive constructs like parDist (n-2) n can give better performance on large architectures, e.g. parMapIntervals on 32 and 64 cores.
5.5. Performance Variability

Finally, we investigate the performance variability induced by each construct. The variability reflects how susceptible the work distribution policy of each construct is to scheduling accidents. We make the following observation, taking \texttt{parMapIntervals} as an example for discussion: (1) The architecture-aware constructs have far less variation than \texttt{par}, with a range of 121s and a standard deviation (sd.) of 33s. (2) \texttt{parDist (n-1) n} and \texttt{parDist (n-2) n} have the least variation (range 13.7s and 10.2s, sd.s 4.3s and 3.2s). It seems that having multiple levels available enables the runtime system to ameliorate scheduling accidents. (3) Of the architecture-aware constructs, \texttt{parAtLeast} has the worst performance.

5.6. High-level Programming Performance

This section evaluates the high level abstraction strategies and skeletons developed in Section 4 using four programs. The \texttt{sumEuler} is similar to \texttt{parMapList} program except it uses the \texttt{parListLevel} strategy instead of \texttt{parMapLevel} skeleton. \texttt{Fib} computes the Fibonacci number using divide-and-conquer parallelism. \texttt{Coins} computes the number of ways of paying a given value from a given set of coins using divide-and-conquer parallelism. Finally, \texttt{Queens} solves the n-queens problem using divide-and-conquer parallelism with an explicit threshold.

<table>
<thead>
<tr>
<th>Program</th>
<th>Runtime (s)</th>
<th>Speedup</th>
<th>Exchanged Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
<td>Arch. Aware</td>
<td>Original</td>
</tr>
<tr>
<td>SumEuler</td>
<td>239.43</td>
<td>-32.97</td>
<td>9.38</td>
</tr>
<tr>
<td>Coins</td>
<td>171.60</td>
<td>-40.60</td>
<td>2.40</td>
</tr>
<tr>
<td>Queens</td>
<td>107.92</td>
<td>-7.85</td>
<td>3.01</td>
</tr>
<tr>
<td>Fib</td>
<td>52.24</td>
<td>-24.39</td>
<td>6.46</td>
</tr>
<tr>
<td>Geom. Mean</td>
<td>-27.42</td>
<td>4.57</td>
<td><strong>6.30</strong></td>
</tr>
</tbody>
</table>

Table 2 reports the speedups, runtimes and number of exchanged messages of the programs on 16 cores with the original strategies and new architecture-aware strategies. The comparison of exchanged messages demonstrates the new approach reduces the communication overhead substantially and thus improves the performance by a mean factor of 1.38 (6.30/4.57).

6. Related Work

Shared memory architectures are sufficiently simple that parallel Haskell does not need architecture awareness to preserve data locality, and the current multicore support in GHC does provide it [9]. The Eden distributed memory parallel Haskell has no architecture awareness. Processes are automatically placed by the runtime system [10]. Other distributed memory parallel Haskells, like Cloud Haskell explicitly place a process at a named location (node)[11]. Architecture aware GrH occupies a middle ground between explicit placement and implicit placement by specifying placement abstractly.

Outside the functional language community, some recent parallel languages are architecture-aware. For example, X10 improves data locality by integrating new constructs: activity, places, regions and distributions [12]. The case study presented in [13] discusses the OpenCL’s performance portability on multicore platform, investigating different strategies to preserve portability. The mechanisms applied in OpenCL approach is similar to the mechanism applied in GrH.

7. Conclusion

We propose four new architecture-aware constructs for GrH on hierarchical architectures. The constructs are implemented using a single \texttt{parDist} primitive in the GUM runtime system. The constructs aid performance portability...
by exposing a virtual, rather than physical, architecture, and minimising prescription. The results obtained for the architecture-aware constructs are consistent for all three programming paradigms investigated. Every architecture-aware construct consistently delivers better speedup and scalability than par with dramatically reduced variability. In summary, we recommend using both parDist and parExact primitives: parDist for expressive power and parExact for simplicity and performance.

Section 4 discusses developing a high-level architecture-aware programming abstractions that abstract over the new constructs. In particular we are developing architecture-aware evaluation strategies and architecture-aware skeletons. Some of the new abstractions, like findLevel and parMapLevel are exploited in the programs measured in Section 5. The new abstractions deliver performance benefits, for example the mean improvement in speedup is 1.38 for the programs measured in Table 2.

There are two open issues with the architecture-aware programming model proposed. One is to identify the thread granularity of tasks. For some programs, like our benchmarks, this may be readily apparent, but some form of resource analysis would be a more general solution. Indeed there are emerging analysis for such hierarchical architectures, like Multi-BSP [14] or SGL [15]. The second issue is how to decide at what level(s) in the architecture to execute a task of a given thread granularity, as discussed in Section 4.1.

The benchmarks were measured only on one hierarchical architecture. Future work would include investigating the model on a range of other hierarchical architectures including larger scale architectures. In particular, GPUs (Graphics Processing Units) are becoming increasingly available for general purpose programming.

Acknowledgements

Thanks to Greg Michaelson for his valuable input. This work has been supported by the European Union grants RII3-CT-2005-026133 SCIEnce: Symbolic Computing Infrastructure in Europe, IST-2011-287510 RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software, and by the UKs Engineering and Physical Sciences Research Council grant EP/G055181/1 HPC-GAP: High Performance Computational Algebra and Discrete Mathematics.

References