A Recursive Data-Driven Approach to Programming Multicore Systems

Rebecca Collins and Luca P. Carloni
Technical Report CUCS-046-07
Department of Computer Science
Columbia University
1214 Amsterdam Ave, MailCode 0401
New York, NY, 10027
December 2007
[rlc2119,luca]@cs.columbia.edu

***Abstract*—In this paper, we propose a method to program divide-and-conquer problems on multicore systems that is based on a data-driven recursive programming model. Data intensive programs are difficult to program on multicore architectures because they require efficient utilization of inter-core communication. Models for programming multicore systems available today generally lack the ability to automatically extract concurrency from a sequential style program and map concurrent tasks to efficiently leverage data and temporal locality. For divide-and-conquer algorithms, a recursive programming model can address both of these problems. Furthermore, since a recursive function has the same behavior patterns at all granularities of a problem, the same recursive model can be used to implement a multicore program at all of its levels: 1. the operations of a single core, 2. how to distribute tasks among several cores, and 3. in what order to schedule tasks on a multicore system when it is not possible to schedule all of the tasks at the same time. We present a novel selective execution technique that can enable automatic parallelization and task mapping of a recursive program onto a multicore system. To verify the practicality of this approach, we perform a case-study of bitonic sort on the Cell BE processor.**

I. Introduction

The performance of single core architectures is not keeping up with Moore's Law. Even though more transistors fit on a chip with each new process generation, there are limits on the amount of instruction level parallelism that can be extracted from a sequential program, and contrary to trends thus far in microprocessor design, the technique of extending pipelines to increase ILP is yielding diminishing returns [9]. If it is difficult to make a larger processor do two times the amount of work as a smaller processor in the same amount of time, why not use two small processors to do twice as much work in the same time as one small processor? Unfortunately, while having two processors can allow one to compute two independent jobs in half the time, it doesn't always allow one to compute one big task twice as fast. Individual programs almost always have data dependencies that force some of the operations to be ordered with respect to each other. Meanwhile, conventional programming models were designed for single CPU systems and either assume sequential execution, or require the programmer to explicitly distinguish concurrent tasks (for example, with POSIX threads). The holy grail of multicore programming is a tool that takes a sequential program and automatically transforms it into an efficient parallel program. Many paradigms for programming multicores can handle the case where there are few interdependencies very well. Parallelization is achieved simply by partitioning the data set and moving the partitions to separate cores. Programming these multicore systems in a way that can make one big task twice as fast (or at least significantly faster), however, is a challenge.

In this paper we introduce a new model of computation for multicores that is based on a recursive data-driven view of the application. Our work focuses on applications that can be defined with *divide-and-conquer* style recursive programs. Our model is driven by the data of the application. Given a recursive program that divides the data with each recursive step, efficient task mapping and scheduling become intuitive. If one pictures the execution of a recursive function as a tree of procedure calls, the data assigned to sibling procedures are more closely coupled than those of cousins. Thus, the decision to co-schedule two tasks can be based on their relationship in the tree - i.e. how tightly coupled their data is. Moreover, the cores can organize themselves dynamically based on how the data is distributed among them. We accomplish this by separating the view of the global data from the actual data itself so that each core recurses on the data structures of the abstract global data set, but selectively executes code that alters the actual data. Each core dynamically determines which code it should execute based on the data that is present locally.

Related Works. Models of computation for multicore programming can be characterized according to what is asked of the programmer and what is asked of the backend (ie. libraries and compiler).

The *data-parallel programming model* can be used when the same operation can be applied to different data independently and it works well with graphical processing units since they typically have a large number of cores, but not a very high-bandwidth intercommunication system [12]. In a data-parallel model, the programmer can express concurrency by defining functions over vectors where a function at an element of the vector does not depend on the values of any of the other vector elements. Data-parallel computing does well with applications that require little inter-task communication. Any overall task

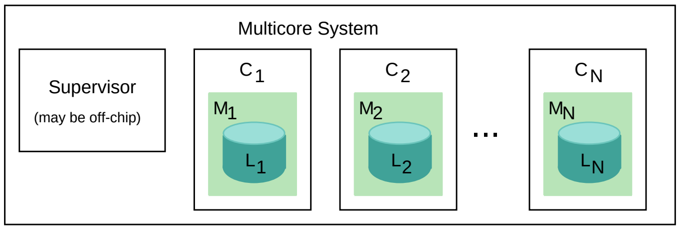


Fig. 1. Abstract Model of a Multicore System.

that can be broken up into many small independent tasks will work. In this model, the programmer must explicitly separate the application into independent pieces, and the backend will manage data transfer and task mapping. However, it is not obvious how one could model data dependencies between separate data-parallel functions.

MapReduce is a programmed model used for handling reduction applications on very large data sets [3], [2]. The *MapReduce* model works in two steps. In the "Map" step, each input token is mapped to a key-value pair. In the "Reduce" step, all of the key-value pairs generated in the Map step are partitioned according to their keys and their values are sent to a reduction function. The programmer provides a function for mapping a single input instance to a key-value pair and reduction functions that handle all of the values matched to specific keys; the backend handles all of the data movement. In multicore systems, *MapReduce* has been used with shared memory systems.

With the *streaming* model of computation [10] a program is defined as a series of filters, where each filter is a unit of computation. Filters have input and output channels and can be pipelined with other filters to build complex operations. In this model the programmer explicitly defines a data flow where different cores will be responsible for different operations. The temporal flow of the data gives good hints about which filters are dependent on each other.

The idea of parallelizing a divide-and-conquer problem according to a recursive function definition has been proposed for Symmetric MultiProcessor (SMP) Architecture from a compiler-oriented perspective, and shown to have good performance [7]. In this approach, the compiler automatically detects data independence in the recursive calls.

The goals of our work differ from those of previous works in two ways. First, we aim to develop a model primarily for applications with strong data dependencies that require communication between concurrent tasks. We prefer systems whose cores have private local memories rather than globally shared memories; our motivation is that when the number of cores scales to the hundreds or even thousands in the future [5], architectures based on globally shared memory will be difficult to sustain. Our methodology is data-driven; it requires that the programmer explicitly partition the data as it is divide-and-conquered through recursive programming, but does not require any concurrent programming beyond this. Then, as the partitioned data is distributed among cores, each core runs the same recursive program, but a core will only follow the recursive paths that overlap with that core's local data.

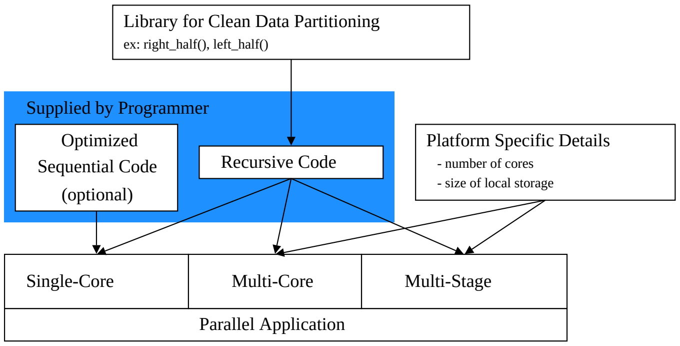


Fig. 2. Software Development Flow

II. Abstract Model of a Multicore System

Before presenting the specifics of our approach, we define how we abstract a multicore system (Fig. 1). A multicore system is made up of N cores that can work together plus 1 *supervising core*. These N + 1 cores may be homogeneous, or they may be different like in the IBM Cell BroadBand Engine [4]. We label the cores C_i, where $1 \leq i \leq N$. Each core C_i has a store of local memory whose capacity is denoted M_i. Together all of the local memories are combined to form the aggregate local memory, $AM = \bigcup_{\forall i} M_i$. The main memory of the system is located off chip, separate from the local memories of the cores.

In typical sequential programming models, the data itself is abstract from the memory. We also keep the notion of data separate from memory and denote the overall data of the program as D. The data stored in a single core C_i's local memory M_i, is labeled L_i, while AL is the data stored in the aggregate memory AM.

III. Method

Figure 2 shows the design flow for developing an application with the recursive data-driven approach. The programmer provides a recursive implementation of the application. The implementation may have a standard sequential form except that when the recursive calls are made, the data must be cleanly partitioned. For example, if the data is in an array format, instead of recursing using C-style pointers, the programmer will use functions like *right_half()* or *left_half()* that will pass the right and left halves of the array but will also keep track of where the data overlaps and how it fits together in the overall data set.

The parallel implementation of the application has three parts, labeled "Single-Core", "Multi-Core", "Multi-Stage" in Figure 2. The single-core part of the application corresponds to a single core executing on data contained in its local memory. The multi-core part corresponds to a multiple cores executing on a set of data that fits within their combined local memories. The multi-stage part corresponds to a task that operates over a set of data too large to fit into the combined local memories of the available cores and so the task must be broken up into smaller pieces that will fit and computed in several stages.

The easiest step of converting the recursive code to the parallel code is creating the single-core code. Since execution on a single core is sequential, the sequential recursive code can be copied directly. For performance, the programmer may also provide a non-recursive implementation optimized for the desired architecture. However, this optimization step is not mandatory for the correctness of our approach.

The challenge in creating multi-core and multi-stage code is that the data will not all fit necessarily into one local memory, and the code must manage how the data is broken up. In addition to the recursive code, for these parts we also add data structures to handle the difference between *global* data and *local* data, but from two different perspectives. In the multi-core code, the global data represents the data that is stored in the combined local memories of the cores (the aggregate data AL) and on each core C_i the local data represents the data that is resident in that core's local memory, L_i.

The multi-core code will run on each core C_i. Initially, the function call will be made on the aggregate data AL, even though no one core contains all of AL. But at each level of recursion, the overall size of the data is iteratively divided by half into smaller AL'. To complete the multi-core code from the original recursive implementation, recursive calls using the partitioned data set will be augmented with conditional wrappers. Two new functions are introduced in these conditionals: *mydata_intersects()*, which is true if $AL' \cup L \neq \emptyset$, and *mydata_contains()*, which is true if $AL' \cap L = L$. Before each recursive call, *mydata_intersects()* and *mydata_contains()* check whether the data used in the recursive call intersect the core's local data L_i or are contained in L_i. If the local data does not intersect the recursed data, the core will skip this part of the code. If the recursed data contained in the local data, the single-core code will be called. Otherwise, when the local data intersects but does not contain the recursed data, the recursion continues in the multi-core code.

In the multi-stage code, the *global* data represents the overall data of the application, D, and the *local* data represents the data that will fit into the combined local memories of the cores AM – the *global* data of the multi-core code.

Conditional wrappers are introduced around the recursive calls in the multi-stage code that check whether the recursed data will fit into the local memory or not. In this case, we just need a function *will_fit* that checks whether the recursed data D' will fit into AM, $|D'| \leq |AM|$. If it will fit, then the multi-core code is called, otherwise, the recursion continues in the multi-stage code. The multi-stage code also takes into account the size of individual local memories M_i, and whether it is necessary to schedule all N cores on a particular task.

There is an important distinction in the way the multi-core and multi-stage codes are run. The multi-core code is duplicated and run simultaneously on the N cores. However, different paths of the recursion are followed by each core according to which part of the global data they have. The multi-stage code is run by only one supervising core.

IV. Example

In this section, a small example parallel application is built from a sequential recursive program. Consider the problem of vector addition, $C = A + B$. This algorithm can be easily parallelized using existing models, but we use it as a simple instructional example. Figure 3 shows a recursive implementation of vector addition.

```
vector_add(int *C, int *A, int *B, int n)
{
  if(n == 1) {
    C[0] = A[0] + B[1];
  }
  C_LH = left_half(C);
  C_RH = right_half(C);
  // etc.
  vector_add(C_LH,A_LH,B_LH, n/2);
  vector_add(C_RH,A_RH,B_RH, n/2);
}
```

Fig. 3: Recursive Vector Addition.

```
multicore_vector_add(List *C, List *A, List *B,
                     int n)
{
  if(mydata_intersects(A_LH,B_LH and C_LH) {
    if(mydata_contains(A_LH,B_LH and C_LH) {
      singlecore_vector_add(C_LH,A_LH, B_LH,n/2);
    } else {
      multicore_vector_add(C_LH,A_LH,B_LH,n/2);
    }
  }
  if(mydata_intersects(A_RH,B_RH, and C_RH) {
    if(mydata_contains(A_RH,B_RH, and C_RH) {
      singlecore_vector_add(C_RH,A_RH, B_RH,n/2);
    } else {
      multicore_vector_add(C_RH,A_RH,B_RH,n/2);
    }
  }
}
```

Fig. 4: Multi-Core Recursive Vector Addition.

For single-core code, we keep the same function, but for clarity, we rename it `singlecore_vector_add()`. Figure 4 shows how `vector_add()` is converted to `multicore_vector_add()`. The integer arrays are changed to (`List *`) data structures. This change decouples the data M_i from the data view AL' that is used in the recursive calls. Each core executes with a high level view of the aggregate data set AL. However, since only a piece of AL will fit into local memory, each core C_i is restricted by `mydata_intersects()` and `mydata_contains()` to only work on parts of the data that are contained in L_i. The `List` structures have information about the positions of AL' data in AL, but do not contain the actual data. Notice that if a core C_i contains the data from the left half of A, B and C, but not the right half, then it will recurse in the first conditional, but will skip the second completely.

Apart from decoupling the data from the data view, the only other change is that the exit case is removed. In the multi-core code, an exit case is no longer needed because it is guaranteed that when the size of the data becomes small enough, `singlecore_vector_add()` will be called to handle the rest of the work. Since `singlecore_vector_add()` is only called on data present in the local memory, it is easy to plug in a more efficient non-recursive version for better performance.

Figure 5 shows the multi-stage code. This code is very similar to the multi-core code. The `List` structure is over-

```
multistage_vector_add(List *C, List *A, List *B,
                      int n)
{
  if(will_fit(A_LH,B_LH, and C_LH) {
    spawn_multicore_vector_add_threads(
              C_LH,A_LH,B_LH,n/2);
  } else {
    multistage_vector_add(C_LH,A_LH,B_LH,n/2);
  }
  if(will_fit(A_RH,B_RH, and C_RH) {
    spawn_multicore_vector_add_threads(
              C_RH,A_RH,B_RH,n/2);
  } else {
    multistage_vector_add(C_RH,A_RH,B_RH,n/2);
  }
}
```

Fig. 5: Multi-Stage Recursive Vector Addition.

```
sort(int *list, int n, int direction)
{
  if(n==1) return;

  left = left_half(list);
  right = right_half(list);
  sort(left, n, direction);
  sort(right, n, direction*-1);
  merge(left, right, n, direction);

  // sort_2() is the same as sort(), except it
  // skips the sort() calls before merge()
  sort_2(left, n, direction);
  sort_2(right, n, direction);
}

merge(int *left, int *right, int n, int dir)
{
  if(n == 1) {
    if((right[0] - left[0])*dir< 0)
      swap left[0] and right[0]
    return;
  }
  left_of_left = left_half(left);
  right_of_left = right_half(left);
  left_of_right = left_half(right);
  right_of_right = right_half(right);

  merge(left_of_left, left_of_right, n/2, dir);
  merge(right_of_left, right_of_right, n/2, dir);
}
```

Fig. 6: Recursive implementation of Bitonic Sort.

loaded here to represent D rather than AL. The function `spawn_multicore_vector_add_threads()` handles the bookkeeping of managing threads and sending data back and forth to the N cores. For both the multi-stage code and the multi-core code, the first call of the recursive function may also be checked in case `will_fit()` or `mydata_contains()`, respectively, is true for the global data set.

For example, if D is smaller than AM, then it is not necessary to recurse at all, we can just invoke `spawn_multicore_vector_add_threads()` directly.

V. Intercommunication - Bitonic Sort

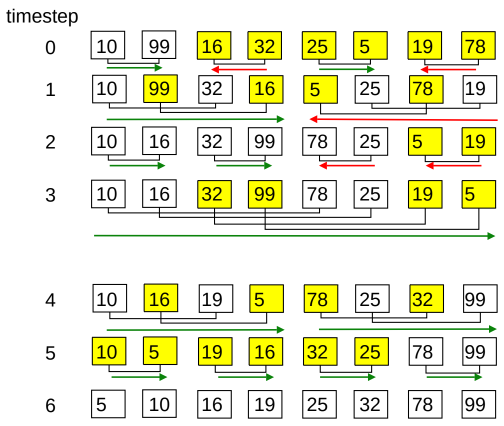
In the vector addition example discussed in Section IV, the parallelization is trivial since no intercommunication is required between the cores. In this section, we describe bitonic sort, the application we use in our experiments. Bitonic sort is a good benchmarking algorithm because its communication pattern is interesting and because hand-optimized implementations have been written on a number of multicore platforms [11], [8].

Bitonic sort is a popular parallel sorting algorithm because the order of compare-and-swap operations is not data dependent. That is, when choosing which elements of the array to compare, the position of the elements in the array matters, but not the values (unlike *quicksort*, for example). In this section, the algorithm and implementation of bitonic sort are explained.

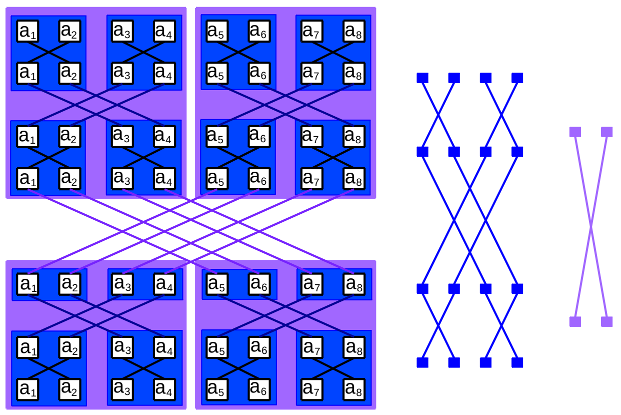
Bitonic sort is a divide-and-conquer algorithm where a list of elements is sorted by first sorting its two halves in opposite directions, and then merging the two halves together. The merge is done by compare-and-swapping the 1^{st} element of the first half to the 1^{st} element of the second half, and then the 2^{nd} element of the first half to the 2^{nd} element of the second half, and so on. The compare-and-swap operations go in the direction that we are sorting the list. After this initial merge (where the elements are $n/2$ elements apart), we repeat the merge $log\ n$ times, but each time we reduce the distance that the elements are apart by half and consider pieces of the list completely separately.

Figure 7(a) shows an example of a list being sorted with bitonic sort. Figure 7(b) shows a picture of how we might map the data dependencies of the arrays elements throughout steps of the algorithm. Depending on how we group the elements, we can look at the algorithm in more or less detail just like a fractal. Notice that at the finest level of granularity, the boxes hold one piece of data. As we increase the granularity so that two or four pieces of data are grouped together, then the structure of the communication remains the same, but overall there are fewer communication paths. In fact, the structure at a course granularity mirrors the fine grained structure of within a local block. In this way, we can look at the algorithm as a divide-and-conquer algorithm where we group locations in the array together according to their location in the hierarchy. So before breaking the computation up to be distributed on multiple cores, we can break up the data into chunks that have good locality. Or, from another point of view, we can break the problem up into chunks that are small enough to fit into one core's local memory or cache.

Consider the recursive implementation of bitonic sort in Figure 6. The structure of the recursive function `sort()` is similar to `vector_add()` in the way that it splits up the input data array with `left_half()` and `right_half()` functions. However, in the middle of `sort()`, there is a call to another recursive function `merge()`. Both `sort()` and `merge()` are divide-and-conquer recursive programs, but they divide their input differently. Figure 8 shows at a high level how data is mapped differently for the two functions. For our preliminary experiments, we handle data passing simply by matching up pairs of cores that together contain the data for `sort()` that should be used by two cores for `merge()` and have the two cores swap data accordingly as shown in the bottom of Figure 8. Internal data swapping has a straightforward pattern for bitonic sort, however the swap patterns may not always be so direct. We can easily detect when swapping should occur - when one recursive program calls another recursive



(a) Example Bitonic Sort



(b) Data Dependencies - Recursive Layers

Fig. 7: Bitonic Sort – (a) An example of how to sort a list using Bitonic Sort. At each stage, the elements of the list are paired up and sorted in the direction shown with the red and green arrows. Elements that will be swapped in the current sorting step are highlighted. (b) The data dependencies for each sorting step are shown. Notice that the position of an element matters, but its value does not, so generic a_i variable names are used.

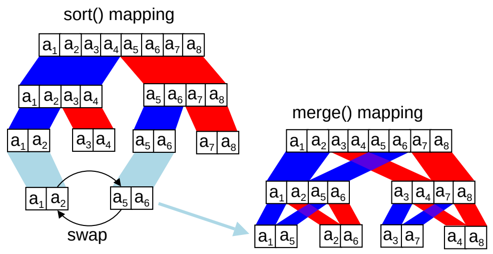


Fig. 8: Data Mapping.

program and the two split data differently. In the most trivial implementation the data could be sent back and forth to the supervisor to convert between the data mappings. However, in the most general case, a tool that can automatically determine how to swap the data between cores on-the-fly is a much more interesting, and likely more efficient, alternative.

VI. Experiments

A recursive data-driven model can be applied to different applications on different platforms. To evaluate performance and scalability of this new model, we implemented Bitonic sort on the Cell BE architecture [4]. As a preliminary exercise, we converted the original recursive function to multi-core and multi-stage code by hand.

Cell BE Architecture. We performed our experiments on a QS20 CellBlade with Cell Software Development Kit 2.0. A QS20 features two Cell BE processor chips together with a 1GB memory. Each Cell chip has $N = 8$ processing cores called SPEs, and one PowerPC core, which is used as the supervising core. Each SPE core has a local memory $M_i = 256KB$ that is used for both data and code. The SPE local memories are not cached. Data is transferred between cores through direct memory access (DMA). The communication network between cores, called the Element Interconnect Bus, is capable of transferring up to 96 bytes per cycle, and the link to main memory is 16 bytes per cycle [1]. Each SPE is a SIMD processor capable of doing the same operation over the elements of a 128-bit wide vector, for example, a vector of four 32-bit integers [6]. We set $L_i = 128KB$ (or $32K$ integers) since bitonic sort only takes input whose size is a power of 2, and we must leave room in M_i for the code and stack. One Cell chip, then has $AL = 1M$ ($256K$ integers). The QS20 has two cell chips, which if used together double the data storage capacity: $AL = 2M$. However, the communication between cores on different chips is not as fast as the communication between cores on the same chip.

Implementation. The core recursive function of our implementation is essentially the same as the code in Figure 6. In our model, the single-core code can be replaced with sequential code to enhance performance. We have three local functions: `sort()`, `merge()`, and `sort_2()`. We replaced the single-core code for `sort()` with a standard *quicksort* implementation [13]. We replaced the recursive single-core code for `merge()` with an iterative alternative. One could also replace `sort_2()` with *quicksort*, but we found that the original recursive version had better performance. However, we did enhance the recursive implementation of `sort_2()` by calling *insertion sort* for small instances to avoid unnecessary overhead from recursive calls at the small scale. For comparisons with single-core systems, we used the same *quicksort* implementation as in our single-core `sort()` code.

Scaling the Input Size. The graph in Figure 9 compares the performance of our bitonic sort implementation with other implementations running on different platforms as the input data size scales from $512K$ to $128M$ integers. Both axes have a logarithmic scale.

CellSort [11] is a hand-optimized bitonic sort implementation for the Cell processor. Our implementation is about 10 times slower than CellSort for smaller input sizes, such as 1M integers, but comes within a factor of 6 in the larger input cases. There are a number of reasons why the recursive model does not come closer to the “ideal” performance of a hand-optimized implementation. First, for single-core computation,

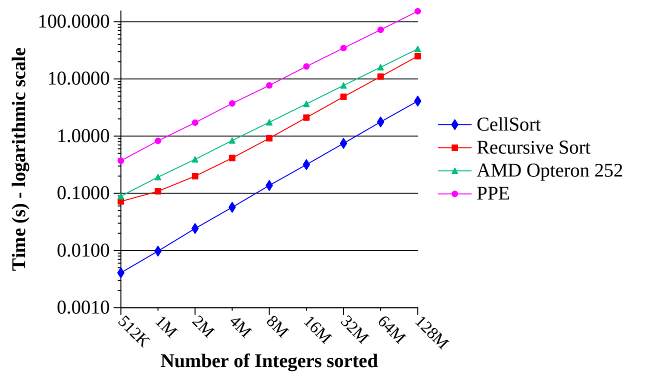


Fig. 9: Scaling the Input Size.

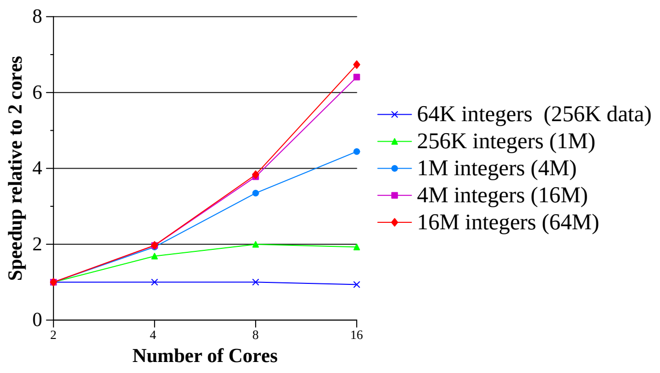


Fig. 10: Scaling the Processing Cores.

CellSort uses a vectorized SIMD bitonic sort implementation that is optimized for the SPE architecture. Our experiments use a standard *quicksort* implementation instead which is about 4 times slower for single-core sorting. Since single-core sort operations grow with the input size, this penalty carries over to larger input sets. In addition, the communication protocols in our implementation are very simple and do not yet contain optimizations such as double buffering. One source of overhead intrinsic to our approach comes from the cost of making recursive function calls. However, since the recursive implementation of bitonic sort cuts the problem size in half with each recursive function call, the number of recursive function calls should grow with the logarithm of the global data size and become less of a penalty as the input size scales up. Moreover, since the cores do not need to use the data during the initial multi-core recursive function calls, the data transfer could be overlapped with these recursive calls in a more optimized version.

Figure 9 also shows the performance times on two single core systems running *quicksort*. The first is an AMD Opteron(tm) Processor 252 with 2.5 GHz , and the second is the PowerPC PPE from the Cell processor. In both cases our recursive implementation on the Cell gives better performance.

Scaling Cores. Perhaps more important than the comparison of multicore to single-core is the performance of the multicore application when the number of cores is greatly increased. The advantage of multicore over single core hinges on the scalability of multicore applications. While there are limits to much faster single core systems can become in the future, the scale of multicore systems is rapidly increasing. Figure 10 shows the speedup observed as the number of cores was increased from 2 to 16 cores. For smaller data sizes such as 64K integers or 256K integers, increasing the number of cores past 4 does not improve performance very much if at all. As the data size scales up, however, the speedup gained from using more cores increases dramatically.

Notice that our programming method allows us to seamlessly scale the deployment of the same bitonic sort code across the 16 SPU and 2 PPE processors that are featured in the two Cell chips hosted on the QS20 CellBlade board.

VII. Concluding Remarks

The recursive model we propose for programming multicore systems is powerful because it expresses concisely (1) data locality - i.e. which parts of the data should be co-located when the overall data is distributed; and (2) temporal locality - if tasks must be scheduled, which tasks should be run at the same time (temporally co-located). In a multicore setting, when recursion is used as a means to reduce the problem size, a single recursive function statement can be used at all of the different levels of the problem - on a single core, on multiple cores, and on multiple cores over multiple stages. Since recursion is naturally hierarchical, it will continue to be an intuitive model even over networks of multicore systems.

To test the performance of our approach, we have implemented bitonic sort on the Cell BE multicore system. We found that at each level of programming, we could reuse the same recursive function which greatly eased development. The recursive implementation of bitonic sort scales with both the input size and the number of cores. The performance came within a factor of 6 of a hand-optimized implementation of the same algorithm. Furthermore, the difference in performance of the recursive implementation and the hand implementation decreased as the problem size scaled up. When scaling the number of cores, we observe that the speedup improved as the program size increased.

Future Work. For this work, we performed the conversion from a recursive program to a parallel recursive program by hand, and have identified several challenging areas that will require future work in developing a tool that can perform this conversion automatically for any general recursive function. When two interleaved recursive functions such as `sort()` and `merge()` have different low level data mappings, the tool must convert between the mappings to direct inter-core data swapping. The partitioning of data is another interesting area of research. In some cases, such as matrix multiplication, the input data should be distinguished from the output data and the input partitions would probably overlap. In other cases, it be best to partition the data in non-contiguous blocks, and for very complicated partitions, the `mydata_intersects()` and `mydata_contains()` functions would also become more complicated.

References

[1] T. W. Ainsworth and T. M. Pinkston. Characterizing the Cell EIB on-chip network. *IEEE Micro*, 27(5):6–14, 2007.

[2] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation,* San Francisco, CA, December 2004.

[3] C. Ranger *et al.* Evaluating MapReduce for multi-core and multiprocessor systems. In *Proc. of the Symposium on High Performance Computer Architecture*, February 2007.

[4] J.A. Kahle *et al.* Introduction to the CELL multiprocessor. *IBM J. Res. Develop.*, 49(4-5):589–604, September 2005.

[5] K. Asanovic *et al.* The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[6] M. Gschwind *et al.* Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[7] M. Gupta *et al.* Automatic parallelization of recursive procedures. *International Journal of Parallel Programming*, 28(6):537–562, 2000.

[8] N. K. Govindaraju *et al.* GPUTeraSort: High performance graphics coprocessor sorting for large database management. In *ACM SIGMOD International Conference on Management of Data*, Chicago, United States, June 2006.

[9] V. Agarwal *et al.* Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proc. Annual International Symposium on Computer Architecture*, pages 248–259, 2000.

[10] W. Thies *et al.* StreamIt: A compiler for streaming applications, December 2001. MIT-LCS Technical Memo TM-622, Cambridge, MA.

[11] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellSort: High performance sorting on the Cell processor. In *Very Large Data Bases Conference (VLDB)*, Vienna, Austria, September 2007.

[12] M. D. McCool. Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. In *GSPx Multicore Applications Conference*, Santa Clara, October 2006.

[13] M. A. Weiss. *Data structures and algorithm analysis in C (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.