A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures

Josep M. Perez [#*1], Rosa M. Badia [#2] and Jesus Labarta [#*3]

[#] *Computational Sciences, Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS)*
Nexus II, C. Jordi Girona 29, 08034 Barcelona, Spain
{[1] josep.m.perez, [2] rosa.m.badia, [3] jesus.labarta}@bsc.es

[*] *Departament d'Arquitectura de Computadors (DAC), Universitat Politècnica de Catalunya (UPC)*
D6 Campus Nord, C. Jordi Girona 1-3, 08034 Barcelona, Spain

***Abstract*—Parallel programming on SMP and multi-core architectures is hard. In this paper we present a programming model for those environments based on automatic function level parallelism that strives to be easy, flexible, portable, and performant. Its main trait is its ability to exploit task level parallelism by analyzing task dependencies at run time. We present the programming environment in the context of algorithms from several domains and pinpoint its benefits compared to other approaches. We discuss its execution model and its scheduler. Finally we analyze its performance and demonstrate that it offers reasonable performance without tuning, and that it can rival highly tuned libraries with minimal tuning effort.**

I. Introduction

Current chip fabrication technologies allow to place several million transistors in a chip, enabling more complex designs each time. However, there are several issues that discourage the design of more complex uniprocessors: the increase in heat generation, the diminishing instruction-level parallelism gains, almost unchanged memory latency, the inherent complexity of designing a single core with a large number of transistors and the economical costs derived of this design. For these reasons, the current trend on chip manufacturing is to place multiple slower processor cores (multi-core) on a chip [1].

As [2] describes, in earlier times performance improvements have often been achieved by simply running applications on new generations of processors with minimal additional programming effort. While current chips have up to 8 cores, this trend may lead in the future to chips with as much as 1000 cores (many-cores). Current programming methodologies will have to drastically change as just recompiling and running the current sequential programs will no longer work. Applications are now being required to harness much higher degrees of parallelism in order to exploit the available hardware and to satisfy their growing demand for computing power. This is seen by many as a real revolution in computing.

Examples of current multi-core chips are several quad-core processors like the AMD K10 Barcelona and the Intel Nehalem. More challenging architectures are for example the Niagara-II by Sun with eight cores, each of them being able to handle eight threads and the Power7 up to 8 cores, each capable of running 4 simultaneous threads.

With such a perspective, the availability of suitable programming environments (compilers, communication libraries, and tools) offering a human-centric approach to exploit parallelism will become essential for the programming productivity of multi-core systems.

In this paper, we present SMP superscalar (SMPSs), a programming environment focused on the ease of programming, portability and flexibility that is based on Cell superscalar (CellSs) [3], [4]. While CellSs is tailored for the Cell/B.E. processor, the solution we present is tailored for multi-cores and Symmetric Multiprocessors (SMP) in general.

Our motivation is to offer a simple programming model, based on the sequential programming flow, but that can exploit the concurrency of the underlying hardware by means of automatic parallelization at runtime. The *same* C sequential code can be compiled with a regular compiler and run sequentially on a single-processor machine; or can be compiled with the SMPSs/CellSs compiler and linked with its runtime library, and run in parallel exploiting the underlying hardware in architectures as different as a laptop with two cores, a node of the MareNostrum supercomputer with two dual-core chips, a Power5 with 4 SMT cores, a Cell/B.E. based blade with up to 18 cores, or an SMP SGI Altix machine with 32 processors.

Parallelism is achieved through hints given by the programmer in form of pragmas that identify atomic parts of the code that operate over a set of parameters. These parts of the code are encapsulated in the form of functions (called tasks). With these hints, the SMPSs compiler and runtime library build a parallel application that detects the task calls and their interdependencies. A task-graph is dynamically generated, scheduled and run in parallel by using a different number of threads depending on the architecture. To achieve good performance, the runtime requires tasks of a certain granularity (e.i. 250 μs). One approach is to block the data and its operations.

The paper is organized as follows: section II overviews the SMPSs programming model. Section III describes the scheduling strategy implemented by the runtime. Section IV is dedicated to programming with blocks. Section V introduces mechanisms to deal with algorithms that do not adapt well to blocking. Section VI compares the performance of SMPSs

978-1-4244-2640-9/08/$25.00 © 2008 IEEE

with other programming paradigms and parallel libraries. Section VII discusses more in depth the similarities and differences with related programming models. Finally section VIII concludes the paper.

II. Programming Model

An SMPSs program is a sequential program annotated with pragmas that identify functions in the code that are candidates to be run in parallel in the different cores. We call those functions "tasks". The task construct declares that a function is a task and specifies the size of each parameter (if it is missing in the C declaration) and its directionality. The syntax of the task construct is the following:

```
# pragma css task [clause [clause] ...]
function-definition | function-declaration
```

where clause is one of the following:

```
input (parameter-list)
output (parameter-list)
inout (parameter-list)
highpriority
```

The first three clauses are the directionality clauses and indicate whether each parameter is either only read, only written or read and written by the task. The highpriority clause indicates that the task has higher priority at scheduling time.

Parameters in the directionality clauses may optionally have dimension specifiers with the following syntax:

```
identifier [[expr] [[expr] ...]
```

where identifier is the name of a parameter and expr is a C99 expression. This is necessary when the parameter is an array and its size has not been specified in the parameter declaration, since the runtime requires its size for proper operation.

The programming environment consists of a source-to-source compiler and a supporting runtime library. The compiler translates C code with the aforementioned annotations into standard C99 code with calls to the supporting runtime library and compiles it using the platform native compiler.

The runtime takes the memory address, size and directionality of each parameter at each task invocation and uses them to analyze the dependencies between them. Only parameters are checked for dependencies. Whenever the application calls a task, a node in a task graph is added for each task instance and a series of edges indicating their dependencies. At the same time, the runtime also schedules the tasks to the different cores as their input dependencies are satisfied.

In order to reduce dependencies, the SMPSs runtime is capable of renaming the data, leaving only the true dependencies. This is the same technique used by superscalar processors [5] and optimizing compilers [6].

This behavior is applied to all parameters except those of type void *. We call them "opaque pointers" since they pass through the runtime unaltered and are not considered in the task dependency analysis.

Renaming is typically applied whenever an algorithm uses a temporary variable or a work array that is accessed by several tasks. In order to avoid false dependencies on those, most programming paradigms require per-thread copies, either through explicit directives or by coding them by hand. This problem is avoided transparently through automatic renaming.

The SMPSs programming model has the same constructions and semantics as those of CellSs. However, due to the differences in the memory architecture, a shared memory environment simplifies programming applications that operate on flat data structures.

III. Scheduling

One of the main goals when scheduling under SMPSs is to exploit data locality. In that regard the scheduler takes advantage of the graph information in order to schedule dependant tasks sequentially to the same core so that output data is reused immediately.

There are two main ready lists, one for high priority tasks and one for normal priority tasks. Tasks in the high priority list are scheduled as soon as possible independently of any locality consideration. The normal priority list is used by worker threads to gather tasks whenever they are idle.

The main code runs on the main thread and the runtime creates as many worker threads as necessary to fill out the rest of the cores. The main thread analyzes task dependencies as it reaches them and adds them to the task graph. Whenever a task is added without any input dependency, it is moved into the main ready list or the high priority list where it can be scheduled by the worker threads.

Each worker thread has its own ready list that contains tasks whose last input dependency has been removed by that thread. Whenever a thread has finished running a task, it updates the graph and moves all task that have become ready to that thread ready list.

Threads look up ready tasks first in the high priority list. If it is empty, then they look up their own ready list. If they do not succeed, they proceed to check out the main ready list. In case of failure, they proceed to steal work from other threads in creation order starting from the next one.

Threads consume tasks from their own list in LIFO order, they get tasks from the main list in FIFO order, and they steal from other threads in FIFO order. This policy allows them to consume the graph in a pseudo-depth-first order as long as they can can get ready tasks, and steal tasks from other threads in a pseudo-breadth-first order when their own ready lists become empty. Cilk has a very similar policy [7].

Due to renaming, the graph only contains true dependencies. As a consequence, all predecessors of a task are the generators of its input data. By following a depth-first execution, the scheduling algorithm favors running tasks in the threads that have just generated one of its input parameters.

The design also tries to keep each thread on a different region of the graph to keep them accessing the same data and thus minimize cache coherency overhead. As long as a thread can find ready tasks in the region it is exploring (thread ready list), or there are unexplored zones in the graph (main ready list), it will not steal tasks from other threads and thus keep the working-sets independent. Work-stealing in FIFO order tries to minimize the effect on the cache of the victim thread by

```
for (int i=0; i < N; i++)
    for (int j=0; j < N; j++)
        for (int k=0; k < N; k++)
            sgemm_t(A[i][k], B[k][j], C[i][j]);
```

Fig. 1. Dense hyper-matrix multiplication code.

```
#pragma css task input(a, b) inout(c)
void sgemm_t(float a[M][M], float b[M][M], float c[M][M]);

#pragma css task inout(a)
void spotrf_t(float a[M][M]);

#pragma css task input(a) inout(b)
void strsm_t(float a[M][M], float b[M][M]);

#pragma css task input(a) inout(b)
void ssyrk_t(float a[M][M], float b[M][M]);
```

Fig. 2. Declarations of some of the tasks that will be used in this paper.

choosing the task that has spent most time on the queue and has more probability of having most of its input data already evicted from the cache.

The main ready list is a point of distribution of tasks in areas of the graph that are not being explored.

The main thread also contributes to run tasks. Whenever it reaches a blocking condition (a barrier, a memory limit, or a graph size limit), it behaves as a worker thread until an unblocking condition is reached.

IV. Programming with Blocks

Blocking is one of the techniques used when programming on task-based languages. In many cases, grouping data into blocks leads naturally to grouping operations between blocks into tasks. This methodology usually leads to tasks with good granularity.

Many problems can be decomposed into smaller problems. This is the case of many linear algebra algorithms like the matrix multiplication, the Cholesky factorization or the LU decomposition without pivoting [8], [9], [10]. In those cases, the components may map easily into tasks that operate on parts of the data. A typical case is to use hyper-matrices to decompose a linear algebra algorithm. In the following examples we will use 1-level hyper-matrixes of N by N blocks, each of M by M elements.

Figure 1 shows a simple SMPSs implementation of the matrix multiplication using hyper-matrices. A, B, and C are N by N hyper-matrices where each position is a pointer to a block of M by M elements. A and B are the input matrices while C is the matrix where the result is stored.

Function sgemm_t is actually a task and its declaration is shown in figure 2 together with some of the tasks that will be used throughout this paper. The pragma css task annotation indicates that the following function is a task. The input, output, and inout clauses indicate which parameters the task reads, writes, or both.

The code generates N^3 tasks arranged as N^2 chains of N tasks. Note that any ordering of the three nested loops produces correct results. The programmer does not have to take care of

```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        for (int k=0; k<N; k++)
            if (A[i][k] && B[k][j]) {
                if (C[i][j] == NULL) C[i][j] = alloc_block();
                sgemm_t(A[i][k], B[k][j], C[i][j]);
            }
```

Fig. 3. Sparse hyper-matrix multiplication code.

```
for (int j = 0; j<N; j++) {
    for (int k = 0; k<j; k++)
        for (int i = j+1; i<N; i++)
            sgemm_t(A[i][k], A[j][k], A[i][j]);

    for (int i = 0; i<j; i++)
        ssyrk_t(A[j][i], A[j][j]);

    spotrf_t(A[j][j]);

    for (int i = j+1; i<N; i++)
        strsm_t(A[j][j], A[i][j]);
}
```

Fig. 4. Left-looking in-place Cholesky decomposition code for dense hyper-matrices.

what is the best task order (for locality, parallelism, ...). The runtime reorders the tasks in such a way that they are run in parallel while exploiting the data locality.

The language and the runtime are capable of expressing and handling dependencies that cannot be resolved at compilation time while preserving the simplicity and avoiding elements unrelated to the original programming language. In most cases, converting a dense algorithm into a sparse variant is simple and straightforward. Figure 3 shows the sparse variant of the multiplication code. This code dynamically allocates memory and executes tasks according to the data needs.

Other codes have much more complexity. Figure 4 contains an in-place left-looking implementation of the Cholesky decomposition [11] using a dense hyper-matrix. This code has complex dependencies that make it hard to parallelize under dependency-unaware programming models [12]. However, the algorithm can be expressed in SMPSs with simplicity and with all dependency analysis it taken care by the runtime.

The dependency complexity is high even for hyper-matrices with few blocks. Figure 5 shows the task graph that would be generated with 6 by 6 hyper-matrices. Each node corresponds to a task that is called by the program and is numbered according to its invocation order. Colors indicate the task type and edges indicate true dependencies. The graph shows that there is parallelism between parts of the code far away in the sequential execution flow. For instance, after running tasks 1 and 6, the runtime is able to start executing task 51, yet the algorithm generates only 56 tasks.

V. Programming with Flat Data

Many algorithms do not adapt well to blocking. The LU decomposition [11] is an example of this class of algorithms. It is usually implemented as an in-place algorithm that receives a two dimensional matrix as input and performs several operations that overwrite its contents. In order to increase the

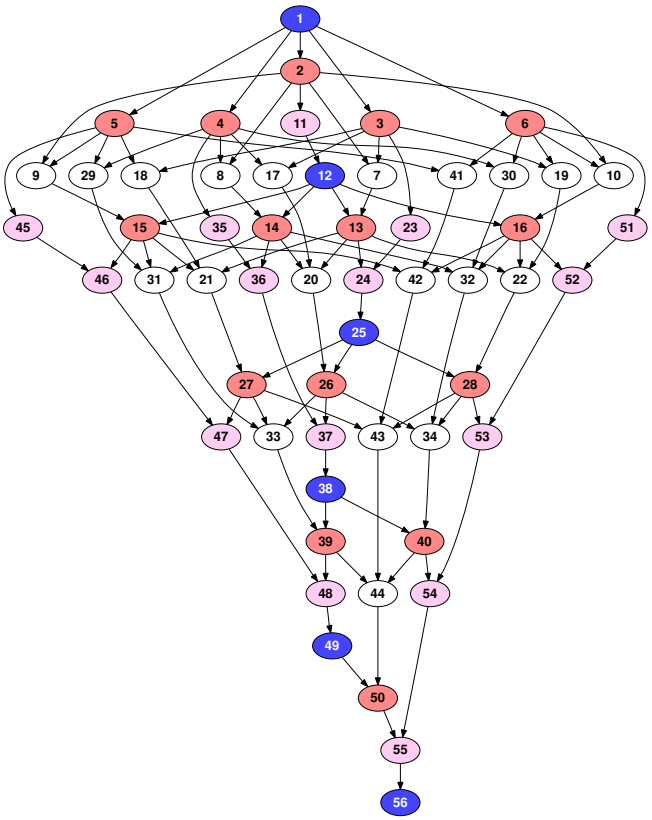

Fig. 5. Task dependency graph created by a 6 by 6 block Cholesky.

numerical stability, the algorithm includes pivoting operations that consist in swapping columns and swapping rows. Those two operations make it hard to block.

Some other algorithms adapt well to blocking, but require accessing the same data several times but each time with a different block size or different memory organization. This is the case of mergesort, where an array is accessed repeatedly but with different and overlapping block sizes.

These two kinds of algorithms motivate the need for a way to specify non contiguous array accesses.

A. Language Extensions

Given an N-dimensional array A with dimensions $d_1, d_2, ...d_N$, we define an array region R from A as a list of pairs $\{p_1, p_2, ...p_N\}$ such that each pair $p_j = (l_j, u_j)$ specifies a lower bound l_j and an upper bound u_j on the corresponding dimension j. R thus, represents all the elements from A with indices $i_1, i_2, ...i_N$ such that each index i_j is within its corresponding bounds inclusively $l_j \leq i_j \leq u_j$ (see figure 6).

The SMPSs syntax can be extended to incorporate array regions for specifying which parts of an array are accessed by a task. Array region specifiers may appear inside the directionality clauses after the (optional) dimension specifiers of each parameter. A region specifier can be specified in three different ways:

$$\{l..u\} \mid \{l{:}L\} \mid \{\}$$

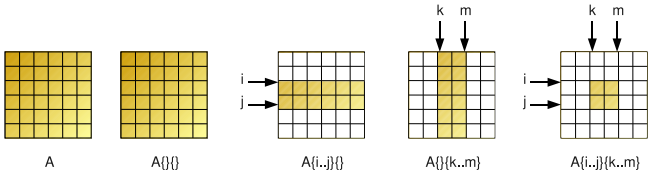

Fig. 6. Elements selected by array region specifiers.

The first syntax specifies the lower and upper bound. The second indicates the lower bound and the length for that dimension. Finally, an empty specifier indicates that the dimension will be fully accessed.

If present, there must be one array specifier per dimension and it is interpreted in the same order as the dimension specifiers. Empty array specifiers indicate that the corresponding dimensions may be used fully.

A single parameter may appear several times in the directionality clauses to indicate that several regions will be accessed. The directionality clause where each region appears on specifies the access type over such region.

Figure 7 shows a variant of mergesort that splits the array into four parts at each recursion step. In this example, array regions are used to specify what parts of the array are accessed by each task.

sort is the recursive function. It receives the data array and an index to the first and last elements that will be sorted. The base case is triggered when the size of the subarray is small. It is solved by invoking the seqquick task that implements a quicksort. The recursive case splits the region into four subregions and calls itself recursively for each of them. Then it merges the first two by calling the seqmerge task and does the same for the last two subregions. The resulting regions are then merged together into the original region of the array.

The seqquick task has one region specifier that indicates that the data array will be read from index i to index j inclusively. The seqmerge task has two different region specifiers, one for the first subarray and another for the second.

B. Representants

Our runtime implementation does not yet include support for array regions. This limitation can be overcome in some cases. However, as a tradeoff, the programmer must take into account dependencies in the code.

A representant is a memory address that represents a possibly non-contiguous collection of memory addresses. Each representant is normally associated to an opaque pointer that is used by the tasks to access the actual data. Pointers with type void * are opaque to the runtime and are passed directly to the tasks skipping any dependency analysis.

Representants can be used to mimic part of the behavior of a region-aware implementation. If the array regions are non-overlapping, it is sufficient to have one representant per array region and an opaque pointer to the array.

By projecting region accesses on their representants, a programmer may introduce back the missing dependency

```
#pragma css task input(data{i1..j1}, data{i2..j2}, i1, j1, i2, j2) \
    output (dest{i1..j2})
void seqmerge (ELM data[N], long i1, long j1, long i2, long j2,
    ELM dest[N]);

#pragma css task inout (data{i..j}) input (i, j)
void seqquick (ELM data[N], long i, long j);

void sort (ELM data[N], long i, long j) {
    ...
    if (size < QUICKSIZE) {
        seqquick (data, i, j);
    } else {
        quarter = size / 4;
        i1 = i; j1 = i+quarter-1;
        i2 = i+quarter; j2 = i+2*quarter-1;
        i3 = i+2*quarter; j3 = i+3*quarter-1;
        i4 = i+3*quarter; j4 = j;

        sort(data, i1, j1);
        sort(data, i2, j2);
        sort(data, i3, j3);
        sort(data, i4, j4);

        seqmerge(data, i1, j1, i2, j2, tmp);
        seqmerge(data, i3, j3, i4, j4, tmp);

        seqmerge(tmp, i1, j2, i3, j4, data);
    }
}
```

Fig. 7. Mergesort code that splits in 4 subarrays each time using array regions.

information. That is, if the region is read, written or both, then by passing the representant as input, output, or inout to the task, an equivalent dependency can be introduced.

However, since renaming is automatic and transparent to the program, representants cannot be reliably used if there are false dependencies between the represented data.

VI. Experimental Results

In this section we evaluate our current implementation against other solutions. It is available in our web page[1] and runs on Linux IA-32, AMD-64, POWER and IA-64. It is composed of a source-to-source compiler and a runtime library.

The performance measurements have been obtained on an SGI Altix computer with 128 cores and 512 GB of memory. It is composed of 32 memory nodes, each with 2 dual core 1.6 GHz Itanium2 processors. The memory nodes are interconnected through a NUMA link. Tests have been run inside a cpuset of 32 cores on 8 nodes with all memory pages bound to those nodes.

Our experiments evaluate the performance of the following algorithms: Cholesky decomposition, matrix multiplication, Strassen, Multisort and N Queens. For the first three algorithms we have implemented the tasks using highly tuned BLAS [13] libraries that already provide high performance implementations of matrix operations. In all cases we used non-threaded Goto BLAS 1.20 [14] and non-threaded MKL 9.1.

[1] http://www.bsc.es/smpsuperscalar

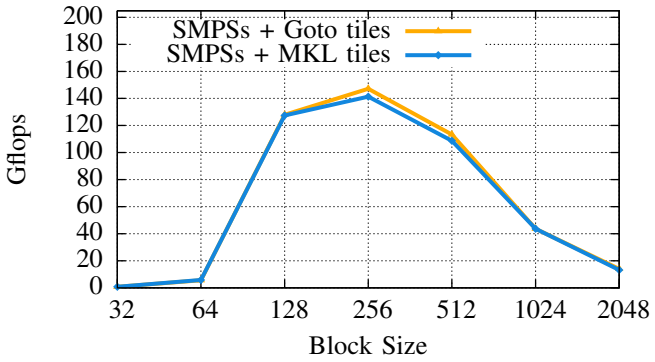

Fig. 8. Performance of Cholesky on the Altix with 32 cores using matrices of 8192x8192 single precision floats and varying the block size.

The Multisort and N Queens algorithms have been adapted from the code examples of the Cilk 5 distribution.

We have run each of these algorithms with 32 threads and a range of block sizes and selected the best performing one. Small block sizes provide lots of parallelism but have high runtime overhead. Conversely, big blocks have reduced runtime overhead but also limited parallelism.

Figure 8 shows the performance of Cholesky algorithm with different block sizes. The execution has been performed on an 8192x8192 single precision matrix. Tasks have been implemented as calls to Goto BLAS (SMPSs + Goto tiles) and Intel MKL (SMPSs + MKL tiles). The top of the chart corresponds to 204.8 Gflops, the theoretical peak with 32 cores.

Note that blocks ranging from 128x128 to 512x512, which would be reasonable candidates before doing any tuning, achieve reasonable performance. With smaller blocks, the amount of per task computation is small compared to the overhead of managing so many tasks (374,272 tasks for Cholesky with 32x32 element blocks, 49,920 with 64x64 blocks). With block sizes from 128x128 to 512x512, there is a tradeoff between parallelism and runtime overhead. With bigger blocks, the performance of Cholesky drops considerably due to decreased parallelism.

Similar experiments have been performed for the rest of the algorithms. Most experiments have also been performed on an SMT 2-way dual core Power5, a 2-way dual core PowerPC 970MP and on an Core Duo processor and they have generated similar results. For space considerations those results have been omitted.

A. Cholesky

In order to evaluate the scalability, we have run an implementation of the Cholesky algorithm and compared it to the parallel versions provided by Goto BLAS and Intel MKL. Since the BLAS implementations operate on a flat matrix, to make the comparison fair, we have implemented the SMPSs version also on a flat matrix.

```
for (int j = 0; j<N; j++) {
    for (int k = 0; k<j; k++)
        for (int i = j+1; i<N; i++) {
            get_block_once(i, k, Aflat, A);
            get_block_once(j, k, Aflat, A);
            get_block_once(i, j, Aflat, A);
            sgemm_t(A[i][k], A[j][k], A[i][j]);
        }

    for (int i = 0; i<j; i++) {
        get_block_once(j, i, Aflat, A);
        get_block_once(j, j, Aflat, A);
        ssyrk_t(A[j][i], A[j][j]);
    }

    get_block_once(j, j, Aflat, A);
    spotrf_t(A[j][j]);

    for (int i = j+1; i<N; i++) {
        get_block_once(i, j, Aflat, A);
        strsm_t(A[j][j], A[i][j]);
    }
}

for (int i = 0; i<N; i++)
    for (int j = 0; j<N; j++)
        if (A[i][j]) put_block(i, j, A[i][j], Aflat);
```

Fig. 9. Left-looking Cholesky decomposition code with on-demand hyper-matrix copies.

Given that our runtime does not support array regions and since there are no dependencies outside of the main algorithm we have chosen to implement it using a simple approach. The flat input matrix is copied block by block into an hyper-matrix on an as needed basis.

We have taken the dense hyper-matrix implementation from figure 4 and prior to accessing any block, we have added code to check if the block has already been copied into the hyper-matrix, otherwise, a task is invoked to copy it. At the end of the algorithm we have added a phase to copy back the block of the hyper-matrix into the flat matrix.

Figure 9 shows the resulting code. The functions and tasks to copy blocks between the flat matrix and the hyper-matrix are shown in figure 10. The flat matrix is stored in Aflat and is always passed to the tasks as an opaque pointer. The corresponding hyper-matrix is stored in A.

Figure 11 shows the results. The SMPSs executions use blocks of 256 by 256 elements.

The MKL parallelization does not scale beyond 4 processors and the Goto parallelization does not scale beyond 10. Given the complexity of the dependencies, we suspect their implementations are limited by them. However, SMPSs is capable of scaling up to 32 processors without any noticeable performance loss.

B. Matrix Multiplication

Figure 12 shows the scalability of the matrix multiplication algorithm with on-demand block copies. The code we used is the original matrix multiplication code but with transformations similar to the Cholesky case in order to make the comparison with the multithreaded BLAS implementations fair.

```
#pragma css task input(A, i, j) output(a)
void get_block(int i, int j, void *A, float a[M][M]) {
    float (*Aflat)[N*M] = (float (*)[N*M])A;

    for (int ii=0; ii<M; ii++)
        memcpy(&a[ii][0], &Aflat[i*M+ii][j*M], sizeof(float) * M);
}

#pragma css task input(A, a, i, j)
void put_block(int i, int j, float a[M][M], void *A) {
    float (*Aflat)[N*M] = (float (*)[N*M])A;

    for (int ii=0; ii<M; ii++)
        memcpy(&Aflat[i*M+ii][j*M], &a[ii][0], sizeof(float)*M);
}

void get_block_once(int i, int j, float Aflat[N*M][N*M],
    float (*(*A)[N])[M]) {

    if (A[i][j] == NULL) {
        A[i][j] = alloc_block();
        get_block(i, j, Aflat, A[i][j]);
    }
}
```

Fig. 10. Tasks and functions to block and unblock on-demand.

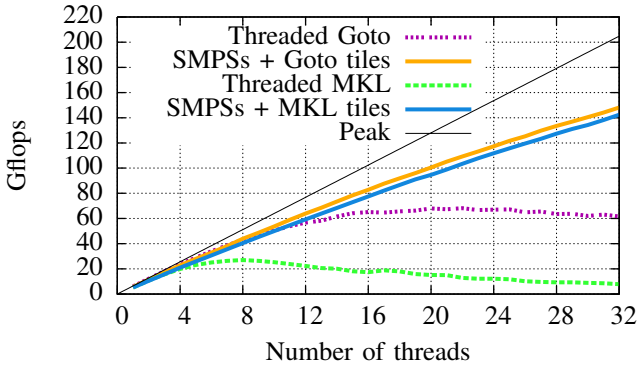

Fig. 11. Performance of Cholesky with matrices of 8192x8192 single precision floats varying the number of processors with SMPSs, Goto BLAS and Intel MKL.

The Goto and the MKL parallelizations are very good and present a smooth response versus the number of threads. In contrast, the SMPSs parallelization exhibits a staircase response due to using of a fixed block size, which leads in some cases to starvation during part of the execution.

Nevertheless, when using a number of threads that does not produce starvation, SMPSs is competitive with multithreaded Goto and MKL. Moreover, with 32 threads it surpasses the MKL parallelization with either MKL and Goto task implementations (tiles).

C. Strassen

The Strassen algorithm performs a matrix multiplication in less than $O(N^3)$ operations [15]. We have run our implementation with hyper-matrices of 8192 by 8129 single precision elements grouped in blocks of 512 by 512. The tasks we used perform block multiplications, additions and substractions.

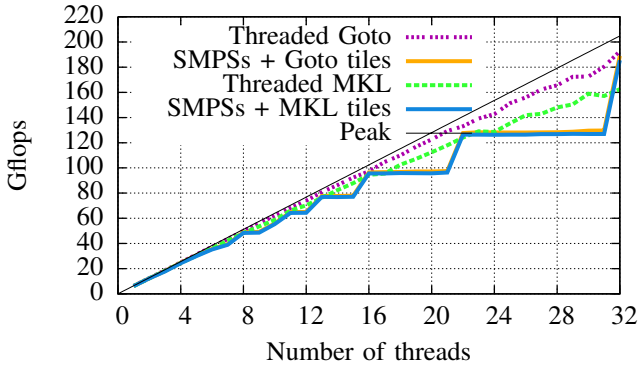

Fig. 12. Performance of matrix multiplication with on-demand block copies with matrices of 8192x8192 single precision floats varying the number of processors.

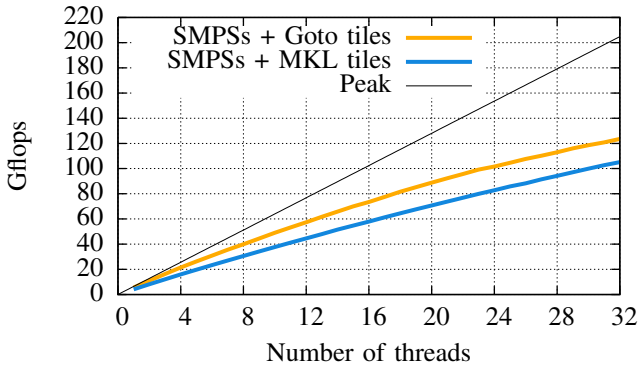

Fig. 13. Performance of the blocked Strassen's algorithm on hyper-matrices of 8192x8192 single precision floats arranged in blocks of 512 by 512 elements varying the number of processors.

While the standard matrix multiplication does not require additional storage, Strassen's algorithm makes heavy usage of temporary matrices, which combined with a recursive implementation, results in an intensive renaming test case.

Figure 13 shows the performance that we achieved. The Gflops figures have been calculated using Strassen's formula from [15]. Compared to the matrix multiplication, Strassen has much smoother response to varying the number of threads, even when using relatively big block sizes. While the multiplication had linear dependencies which combined with big block sizes produced a staircase effect, the Strassen algorithm is more adaptable to the number of threads thanks to a less linearized graph which allows more work-stealing and prevents starvation.

The number of Gflops per second obtained is lower than for the matrix multiplication. There are two main reasons. First, the additional memory allocations required for renaming increase the time spent on the runtime. Second, Strassen trades block multiplications with additions and substractions, which have less arithmetic operations per memory access, thus demanding more memory bandwidth.

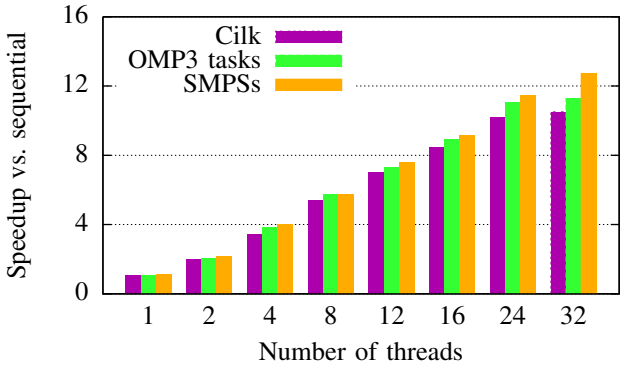

Fig. 14. Performance of multisort varying the number of processors.

D. Multisort

Multisort is a modified implementation of mergesort that uses a divide-and-conquer approach to merge each pair of sorted arrays [16]. The code is similar to that of figure 7, except that the seqmerge task invocations have been replaced by calls to a recursive merge function that ends up calling said task when the operated range is small enough.

The main recursive part uses quicksort to solve the base case and insertion sort for very small regions. The code has been based on example code from the Cilk distribution.

Figure 14 shows the speedup of the Nanos OpenMP 3.0 tasks, Cilk and SMPSs implementations compared to the sequential implementation. All three versions scale similarly, with SMPSs having slightly better performance than the others.

E. N Queens

This benchmark solves the n-queens problem, whose objective is to find a distribution of N queens on an N by N board such that no pair of queens attacks each other. The code is based on example code from the Cilk distribution and is implemented using recursion. At each recursion step, a position of the solution array is tried with all different possibilities.

The OpenMP tasking version has as task the body of the queens function. To allow certain amount of task granularity, the last 4 levels of recursion are computed by a sequential task that does not get decomposed.

The SMPSs version is very similar to the OpenMP version. However, since it does not handle recursive tasks, the queens function is decomposed recursively until the last 4 levels, and those are handled by tasks.

The Cilk version is totally recursive and does not make any depth distinction.

While the sequential version of the program can find all solutions with just one solution array, the OpenMP 3.0 tasking version and the Cilk version cannot. At each nested task entrance the OpenMP tasking version requires allocating a copy of the partial solution array so that tasks at the same

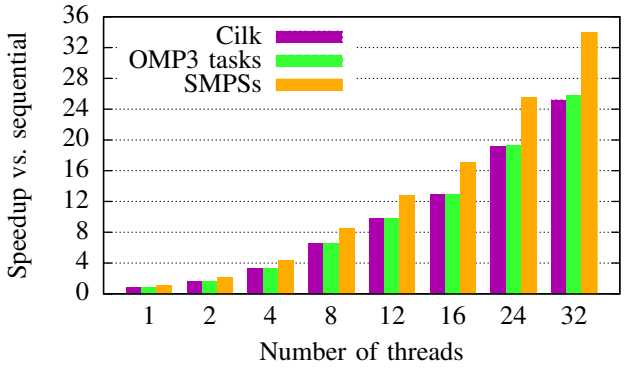

Fig. 15. Performance of N Queens varying the number of processors.

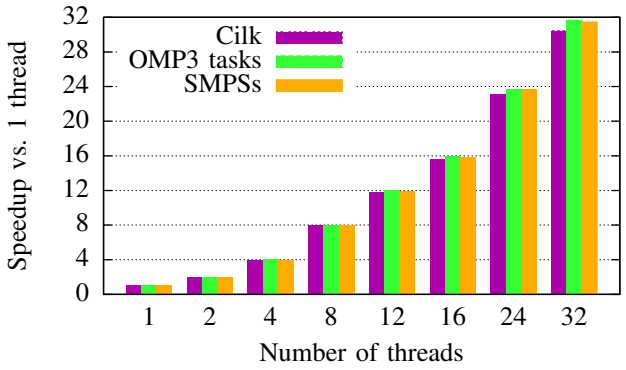

Fig. 16. Scalability of N Queens with array duplications varying the number of processors compared to the same programming model with 1 thread.

recursion level do not overwrite each other's partial solutions. Cilk has exactly the same problem.

Like the sequential version, SMPSs does not require duplicating the partial solution array by hand. The runtime takes care of it by renaming the array as needed. Figure 15 shows the speedup of each implementation versus the sequential execution. SMPSs obtains better performance with 1 thread than the sequential execution. This is due to the runtime realigning data due to renamings and to the increased locality due to the task reordering. This advantage is preserved with more threads.

We discovered that many publications that compare the sequential version with either the Cilk or the OpenMP tasking version do so with a sequential version that performs those array duplications. We consider that a sequential version should not contain artifacts necessary for a parallel paradigm. However, by comparing with such a version, we can infer a measure of their scalability. Figure 16 shows the scalability of each version against the time of a single thread execution under the same programming paradigm.

VII. Related Work

A. CellSs

This work is based on previous work on CellSs [3], [4] and GRID superscalar [17]. While many parts of the infrastructure of CellSs and SMPSs are shared, they are tailored for different environments. CellSs was specifically designed for the Cell/B.E. chip, which is an heterogeneous architecture. This work is tailored for homogeneous architectures with shared memory. While CellSs has a set of strict requirements on memory due to requiring DMA data transfers, this limitation is non-existent on SMPSs.

The scheduling algorithms in both SMPSs and CellSs try to exploit data locality. They take advantage of the graph information in order to schedule dependant tasks together so that output data is reused immediately. However, while CellSs has a centralized scheduler that pre-schedules groups of tasks together, SMPSs has a distributed scheduler that schedules tasks only when they are ready. Moreover, SMPSs has one queue per core to keep related tasks on the same core and performs work-stealing, while CellSs has a unique queue and does not employ work-stealing.

B. OpenMP 3.0

OpenMP 3.0 [18] is incorporating a tasking model based both on the taskqueuing model [19] and dynamic sections [20]. It increases the flexibility of previous versions by allowing it to better accommodate irregular algorithms. Nevertheless, the original task pool proposal does not contemplate dependencies, greatly limiting its effectiveness in case of their existence [12]. OpenMP 3.0 supports nested tasks and balancing those among the cores, while SMPSs treats task calls inside tasks as normal function calls. Tasks with dependencies in OpenMP have been proposed in [21].

C. SuperMatrix

The SuperMatrix [22] approach is very similar in motivation and in technique to both CellSs and SMPSs. In this case, SuperMatrix is focused on linear algebra algorithms expressed in terms of recursive problem decomposition. It provides a library with a set of ready to use linear algebra routines that use tasks internally and an API to build algorithms composed of those routines.

In contrast, SMPSs is designed to be generic by enabling the programmer to define the building blocks (tasks) and thus it is not limited to just one field. It also allows to develop programs in both recursive and iterative ways. In that regard, it is independent of the main control flow of the program.

The SuperMatrix task dependency analysis mechanism is similar to the SMPSs and CellSs one. One notable difference is that SuperMatrix does not support renaming. SMPSs tries to preserve the programming structure and syntax of a normal sequential program, where it is expected that renaming will help removing false dependencies due to temporary variables.

Another difference is that SuperMatrix has a central ready queue and its locality approach is based on assigning each block to one core and run tasks that write to that block

only on the assigned core [23]. This assignment is performed independently of task dependencies. In contrast, SMPSs has several ready queues and follows the graph dependencies in order to exploit data that is hot in cache.

While both SMPSs and CellSs start executing tasks as soon as they enter the graph, SuperMatrix first develops the whole graph, and then stops the main flow execution until the graph has been fully consumed.

Finally, while SuperMatrix is composed of just one library of routines, SMPSs is composed of a set of tools focused on the programmer consisting of a compiler, a standard runtime and a tracing-enabled runtime. The tracing-enabled version records events related to task creation and execution for post-mortem analysis with the Paraver tool [24].

D. Cilk

Cilk [7] is also a task-based multithreaded programming environment. The Cilk programming model is tailored to recursive problem decomposition. In contrast, SMPSs handles calls to tasks from within tasks as normal function calls.

Cilk does not handle task dependencies across tasks in the same recursion level. Moreover, the programmer must place barriers before exiting a task in order to wait for the results of its sibling tasks. This limits the level of parallelism that can be obtained. SMPSs handles data dependencies internally while trying to avoid synchronization points. As a consequence, SMPSs can run in parallel tasks that are distant in the code.

For instance, a real program may perform a Cholesky factorization and use the result in another operation. As the results of the factorization become available, the tasks of the second operation that consume them can be executed, recovering the parallelism lost as the execution reaches the bottom of the Choesky graph (see figure 5).

Similarly to OpenMP, sequential versions of SMPSs programs can be obtained by just compiling the same source under any standard C compiler and the sequential version can be debugged with any standard debugger. The extensions to C added by Cilk are intrusive and do not allow to obtain a sequential version by just ignoring them.

The SMPSs and Cilk scheduling algorithms are similar but they have different motivations. In Cilk work-stealing is done in FIFO order to steal tasks as "big" as possible and reduce the time spent stealing tasks. SMPSs does so to try to operate on a different part of the graph and thus avoid cache conflicts. Another difference is that the SMPSs programming model incorporates the ability to specify that a task has high priority and should be scheduled as soon as it gets ready.

VIII. Conclusions

We have presented a task-based programming environment for SMP and multi-core chips that is easy, flexible and portable. It allows the programmer to forget about data dependencies and instead concentrate on the program itself. It is very powerful in terms of parallelism extraction for both regular and irregular block based algorithms whether the data is structured in blocks or is flat.

Its simplicity has been demonstrated by means of code examples and descriptions of the various approaches to programming with SMPSs. Its renaming capability avoids common code transformations when parallelizing for other programming models, like replicating variables and performing content copies by hand in the original source code.

However, simplicity is not a tradeoff for power as witnessed by the ability to exploit parallelism in the presence of complicated dependencies between distant parts of the code.

A language extension has been proposed in order to help programming applications that operate on data that is not naturally structured in blocks, thus extending the range of applicability.

The SMPSs performance has been analyzed and compared to other programming models and manually tuned parallel libraries. The results have shown the advantages of SMPSs both in terms of programmability and in terms of performance. In those results we have demonstrated that even without tuning, an application under SMPSs can have respectable performance and with tuning it can reach and in some cases outperform highly tuned parallel libraries.

SMPSs is open source software and is available at the SMPSs web page[2].

Acknowledgements

This work has been partially supported by the CICYT under contract TIN2007-60625 and the BSC-IBM Master R&D Collaboration agreement.

We would like to thank Alex Duran, Isaac Jurado and Luis Martinell for their help in the experimental results section. We also would like to thank Eduard Ayguadé for his detailed and insightful comments on a previous manuscript of this paper.

References

[1] D. Geer, "Chip makers turn to multicore processors," *Computer*, vol. 38, no. 5, pp. 11–13, May 2005.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[3] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: a programming model for the Cell BE architecture," in *proceedings of the 2006 ACM/IEEE conference on Supercomputing*, November 2006.

[4] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta, "CellSs: Making it easier to program the Cell Broadband Engine processor," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 593–604, September 2007.

[5] J. E. Smith and G. S. Sohi, "The microarchitecture of superscalar processors," in *Proceedings of the IEEE*, vol. 83, no. 12. IEEE, Dec. 1995, pp. 1609–1624.

[6] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1981, pp. 207–218.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," *SIGPLAN Not.*, vol. 33, no. 5, pp. 212–223, 1998.

[2] http://www.bsc.es/smpsuperscalar

[8] R. C. Agarwal and F. G. Gustavson, "Vector and parallel algorithms for Cholesky factorization on IBM 3090," in *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 1989, pp. 225–233.

[9] B. Kågström, P. Ling, and C. van Loan, "GEMM-based level 3 BLAS: high-performance model implementations and performance evaluation benchmark," *ACM Trans. Math. Softw.*, vol. 24, no. 3, pp. 268–302, 1998.

[10] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," LAPACK Working Note 191, Sep. 2007, arXiv:0709.1272.

[11] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in FORTRAN; The Art of Scientific Computing*. New York, NY, USA: Cambridge University Press, 1993.

[12] A. Duran, J. M. Perez, E. Ayguade, R. M. Badia, and J. Labarta, "Extending the OpenMP tasking model to allow dependent tasks," in *Proceedings of the 4th International Workshop on OpenMP*, West Lafayette, Indiana, USA, May 2008, to be published.

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software*, vol. 5, no. 3, pp. 308–323, 1979.

[14] K. Goto and R. van de Geijn, "High performance implementation of the level-3 BLAS," *ACM Transactions on Mathematical Software*.

[15] V. Strassen, "Gaussian elimination is not optimal." *Numerische Mathematik*, vol. 14, no. 3, pp. 354–356, 1969.

[16] S. G. Akl and N. Santoro, "Optimal parallel merging and sorting without memory conflicts," *IEEE Trans. Comput.*, vol. 36, no. 11, pp. 1367–1369, 1987.

[17] R. M. Badia, J. Labarta, R. Sirvent, J. M. Pérez, J. M. Cela, and R. Grima, "Programming grid applications with GRID superscalar," *Journal of Grid Computing*, vol. 1, no. 2, pp. 151–170, 2003.

[18] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, M. Federico, E. Su, P. Unnikrishnan, and G. Zhang, "A proposal for task parallelism in OpenMP," in *Proceedings of the 3rd International Workshop on OpenMP*, June 2006.

[19] S. Shah, G. Haab, P. Petersen, and J. Throop, "Flexible control structures for parallelism in OpenMP," *Concurrency and Computation: Practice and Experience*, no. 12, pp. 1219–1239, 2000.

[20] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos mercurium: A research compiler for OpenMP," in *Proceedings of the 6th European Workshop on OpenMP*, September 2004, pp. 103–109.

[21] M. Gonzalez, E. Ayguadé, X. Martorell, and J. Labarta, "Exploiting pipelined executions in OpenMP," in *Proceedings of the 32nd Annual International Conference on Parallel Processing (ICPP'03)*, October 2003, pp. 153–160.

[22] E. Chan, F. G. van Zee, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn, "Satisfying your dependencies with SuperMatrix," in *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, K. M. Marcin Paprzycki, Ed. Austin, Texas, USA: IEEE Computer Society, September 2007.

[23] K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson, "An experimental comparison of cache-oblivious and cache-conscious programs," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 93–104.

[24] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, "Dip: A parallel program development environment," in *Proceedings of the 2nd International Euro-Par Conference*, vol. 2, August 1996, pp. 665–674.