Parallelism granules aggregation with the T-system.

Alexander Moskovsky, Vladimir Roganov, and Sergei Abramov

Program System Institute Russian Academy of Science,
Pereslavl-Zalessky, 152020, Yaroslavl Region, Russia
moskov@lcc.chem.msu.ru,var@pereslavl.ru,
abram@botik.ru
http://www.botik.ru/PSI

Abstract. T-system is a tool for parallel computing developed at the PSI RAS. The most recent implementation is available on both Linux and Windows platforms. The paper is dedicated to one of important T-system aspects — ability to change parallelism granule size at runtime. The technique is available, primarily, for recursive programs, but it's possible to extent it to non-recursive ones as well. In the latter case, we employ C++ template"traits"for program transformation. The technique is shown to reduce overhead incurred by runtime support library dramatically.

Key words: T-system, OpenTS, parallel programming, C++, computational clusters, parallel computing

1 Introduction

The building of snow castle is a favorite winter-time game of Russian children. One has to use large snowballs to build one (or two, or three) walls and, may be, a tower. Then the castle is ready to protect it's builders from others in snowball game. If weather is appropriate (just below zero Celsius plus a major snowfall), snow castles mushroom along recreation areas in residential city blocks. One, who ever seen a snow castle know, that it's constructed of large snowballs, not small ones, the size is to be like ones to build a snowman, or even larger. It's just too boring to build snow castle out of small snowballs, while small ones can be used to fill gaps between major ones, level walls and correct imperfectness. And it's not possible to build a snow castle treating snowflakes individually.

As well, the proper choice of granule size is very important for the parallel computation to be effective. If the granules are too large, there may be not enough granules to load all available CPUs. With large granules the cost of scheduling error is larger too: cohesion, caused by assigning tasks to wrong CPU, will last longer. At the same time, if granules are too small, the overhead incurred by runtime system may be too large. In this paper we describe the granule size control techniques, applicable for the context of the T-system approach. In T-system, a potential granule is called"T-Function" and is, actually, a C-functions

that can be computed in parallel. That creates a possible conflict of goals: the "functions" in C/C++ program are to structure source code and make program easier to read, while in T-system they serve as granules of parallelism, which size should be large enough to pay back runtime system overhead. So, the dynamic granules size — aggregation of multiple function calls in single granule — may be very important to improve ease of parallel programming with the T-system.

2 Related work

The ability to dynamically adjust size and number of parallelism granules can be enabled by either well-defined program structure or rigorous approach, based on functional programming and graph reduction. The Open MP [1] may be considered as the most widely used implementation of the first approach: number of threads created is in particular section is defined equal to number of CPUs available. However, Open MP is mostly applied to loop parallelization, when loop iterations have approximately equal CPU instructions. In more sophisticated cases, graph partitioning is widely used in high-performance scientific calculations involving meshes [2].

Much more general approaches exist in the realm of parallel functional programming. "Task inlining"[3], "lazy task creation"[4] and "leapfrogging"[5] has been devised almost two decades ago for Mul-T [3] and Multilisp[6]. In principle, all these techniques are applicable in context of any futures-based system, like it has been recently shown for Java-based system in [7]. A lot of work has been done on granularity control in Glasgow Parallel Haskell both in terms of CPU and memory resources [8, 9]. However, the application of these approaches in high-performance computing requires very tight limits on overhead, added by the mechanism. For the parallel programming environment to be useful, it should not only provide good speedup when running parallel programs, but allow low-level optimization as well. It's well known, that good optimizing compiler may improve speed of application by 100-300% and this requires only applying some optimization switches during compilation. For the majority of platforms, good optimizing compilers are available for C/C++ and Fortran, but not other programming languages. It may also important to allow compiler apply loop optimization techniques, like loop unrolling, vectorization, skewing etc [10], to load CPU pipelines and multiple execution units. The T-system design addresses these issues: we use C++ as a basic language, and allow low-level optimization in parallelism granules, at the same time, C++ inlining, in principle, should allow compiler to optimize loops including "aggregated" granules (see below).

In this paper we focus on implementation of granules aggregation in the context of T-system [11]. T-system provides programming model, which extends C++ language to express parallelism, and runtime support library to enable program execution on multi-cores, SMPs, computational clusters. The T-system enables writing much more compact programs model than traditional Message-passing interface (MPI) libraries. There are many C++ extensions available, designed to provide high-level programming tools [12]. It's interesting to note,

that even original C++ design goals was to support parallelism, but it has been decided later to rely on libraries with that aspect [13]. Rather novel approach relies upon C++ templates to "extend" C++ language for parallelism [14–16]. Due to size limitations of this manuscript, we would like to refrain from comprehensive overview of C++ language extensions. However, we must notice distinctive features of T-system approach:

- "Functional-based" approach to parallelisation (see below)
- Mutliple assigment support for global variables (in T++ language, see below)
- Custom ligtweight thread library (in Open TS)
- Distributed garbage collector

The ability to aggregate granules, specified by a programmer is a distinctive feature as well. Combined with the availability on both Windows and Linux Platform makes T-system convenient tool to a wide community of potential users.

3 OpenTS: T-system implementation

Open TS is the most recent full-scale implementation of T-system approach [11]. OpenTS provides a T++ — a language for parallel programming, which is a seamless extension of C++ language with only 7 keywords:

- **tfun** — a function attribute which should be placed just before the function declaration. A function with the "tfun" attribute is named "T-function", and runtime support system can compute such functions is parallel — in separate threads of execution.
- **tval** — a variable type attribute which enables variables to contain a non-ready value. The variable can be cast to the "original" C++-type variable, which makes the thread of execution suspend until the value becomes ready.
- **tptr** — a T++ analogue of C++ pointers which can hold references to a non-ready value.
- **tout** — a function parameter attribute used to specify parameters whose values are produced by the function. This is a T++ analog of the "by-reference" parameter passing in C++.
- **tdrop** — a T++ -specific macro which makes a variable value ready. It may be very helpful in optimization when it's necessary to make non-ready values ready before the producer function finishes.
- **tct** — an explicit T-context specification. This keyword is used for specification of additional attributes of T-entities.

Generally, Open C++ [17] reflection is used for conversion of the T++ programs to C++ with calls to Open TS runtime support library. The simplest sample program — Fibonacci numbers calculation is presented below.

```
tfun int fib(int n)
{
```

```
    if (n<2) return 1;
    return (fib(n-1)+fib(n-2));
}
tfun int main (int argc, char *argv[])
{
    int n = atoi(argv[1]);
    printf("Fibonacci %d is %d\n",n,(int)fib(n));
    return 0;
}
```

Casting `(int)fib(n)` is necessary to make `main` thread to wait for other threads to complete. Open TS runtime support library relies on MPI for communication in cluster environment, while addtional options are available (PVM, and TCP/IP when MPI is not applicable). Open TS features custom lightweight thread library, which is capable to make up to millions of context switches on a modest CPU. Another important Open TS capability is automatic garbage collection of non-ready values. By the end of 2006, the Windows port has been finished. The "cross-platform" version is available for download at URL `http://www.opents.net`.

4 Granule aggregation in recursive programs

Sometimes, even bantamweight threads are too heavy: the program may be most naturally expressed in terms of functions which take only few CPU instructions to compute. The Fibonacci example above is program of that kind: most of CPU time is spent on thread and non-ready values management by the runtime system, not on summation of integers. One may require programmer to coarsen parallelism grains supplied to system — the simplest solution. Consider the following modification of original Fibonacci code:

```
int cfib (int n) {
    return n < 2 ? n : cfib(n-1) + cfib(n-2);
}

tfun int fib (unsigned n)
{
  if (n < 32) {
    return cfib(n);
  } else {
    return fib(n-1) + fib(n-2);
  }
}

tfun int main (int argc, char* argv[])
{
  int n;
```

```
  if (argc < 2) {
  fprintf(stderr,"Usage: %s <number>\n", argv[0]);
    return -1;
  }
  n = atoi(argv[1]);
  printf("fib(%d) = %d\n", n, (int)fib(n));
  return 0;
}
```

The whole source is obscured a bit, moreover, the summation is replicated in two pieces of program — making in harder to support. The alternative for OpenTS is an implementation of technique, similar to "inline" of MultiLisp [6]. In that case, when a user program is calling a T-function "fib", runtime system may decide don't create any new T-threads, but, instead, evaluate a function, calling it's as ordinary C-function. That reduces parallelism: the runtime system will not be able to make some threads run in parallel. At the same time, it removes much overhead from runtime execution, since there is no need to create extra task object, schedule it and so forth. The benefit in terms of execution time reduction on one CPU is observable for Fibonacci:

Table 1. Exectuion times for calculating 41-st Fibonacci number

Program	Number of threads	Execution time
Fib(41)	535828592	7108.952 sec
Fib(41)-aggregate	8192	5.603 sec
Fib-cilk-5.4.3	n/a	19.7 sec

Here and below, measurements hes been done on dual Athlon MP 1800+ system with 1Gbyte of RAM, only one CPU was used. Program has been built with GNU C++ compiler version 3.2.2 and -O3 optimization flag. Here we applied a simple heuristic: calls, with recursion level deeper than the threshold (namely, 17), are implemented as C-call, not thread-creating calls. For comparison, we present also running time for calculation of the 41-st Fibonacci number with Cilk version 5.4.3, which is approximately 19 seconds. The Cilk [20] is a multi-threading programming environment for symmetric multi-processors (SMPs) and multi-core processors, which won HPC Challenge class 2 (most productivity) [21] award on Supercomputing conference [22] in the year 2006.

The recurtion depth heuristic can be applied for a more sophisticated program: calculating the π number with the numerical integration method (it's concept similar to `sum-tree` test of [4]):

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
```

```
tfun double isum(double begin,
                     double finish,
                     double d) {
    double dl = finish - begin;
    double mid = (begin + finish) / 2;
    if (fabs(dl) > d)
        return isum(begin, mid, d) +
                    isum(mid, finish, d);
    return (double)f(mid) * dl;
}
tfun double f(double x) {
    return 4/(1+x*x);
}
tfun int main(int argc, char* argv[ ]) {
    unsigned long h;
    double a, b, d, sum;

    if (argc < 2) {return 0;}
    a = 0; b = 1; h = atol(argv[1]);
    d = fabs(b - a) / h;
    sum = isum(a, b, d);
    printf("PI is approximately %15.15lf\n", sum);
    return 0;
}
```

One may notice, that only minor changes were necessary to make this program to run in parallel with the Open TS. Without granules aggregation, the overhead, introduced by the T-system would be very large, comparing it with few CPU instructions, which are necessary to calculate the “isum” function. To make this program efficient in that case, it would be necessary to create a loop inside the “isum” function, calculating the “f” multiple times. The T-system runtime with support of recursive granule aggregation is much more forgiving: it even allows placing “tfun” keyword for “f” function which is not practical to calculate in parallel on either cluster or SMP system. Consider the run time measurements: Only subtle differences are observable, one the scale of hundredth of second.

Table 2. Exectuion times for calculating π, 100000000 points

Program	Number of threads	Execution time
Pi — no aggregation	402653184	5589.667
Pi (tfun f)	8192	11.670 sec
Pi (no tfun)	8192	11.543 sec
Pi — C version	1	8.774 sec

C version is produced, removing all T++ keywords from the source code by preprocessor, which result in sequential program.

5 Granules aggregation in "Map" parallel programming template

The "Map" high-level function is widely known concept in functional programming [19]. The "Map" takes two arguments: input set and function, which has to be applied to each element of the input set, producing the output set. Since the operation on elements of the input set are independent from each over, parallelization of "Map" is straightforward. In C++, the high-level function can be implemented with the help of template functions. In C++ Standard Template Library(STL) it's a "transform" template, taking input, output iterators and function. In many cases, "Map" may be substitute for "for" loops. It is also may be beneficial to use "Map" instead of loop, since loop parallelization in plain T++ requires at least two loops instead of one: C++ code:

```
int do_something(int);
...
int res[NMAX]
for (int i=0;i<NMAX;++i)
    res[i]=do_something(x[i]);
```

The equivalent T++ code looks like: T++ code:

```
tfun int do_something(int x);
...
tval int tres[i];
for (int i=0;i<NMAX;++i)
    tres[i]=do_something(x[i]);
for (int i=0;i<NMAX;++i)
    res[i]=tres[i]
```

We have implemented the "Map" template with the C++ language and T-Sim C++ template library. It is based on "futures" [6] approach to parallelization, thus it's compatible with OpenTS in many aspects. Details of this library will be presented elsewhere. For the sake of implementation simplicity, user should supply the "functoid" [18] object to the template. The "Map" based code for an example above may look like the following:

```
int do_something(int);
...
Functoid<do_something> f;
MapD(x,x+NMAX,res,f);
```

The condition of speedup on parallel machine for this program is that the function `do_something` must constitute large enough chunk of work. But, in general

case, it may not be sufficient to pay back amount of time, spent by runtime support library on handling the task and data transmission. That general case may be the simplest for the programmer to implement. However, our "Map" template is capable aggregating individual operations, producing larger grains and reducing the runtime overhead. Currently, programmer should supply an extra parameter to the template, "trait" for granule aggregation. Consider the following fragment for aggregation by the compile-time specified number:

```
MapA<FixedAggregation<100> >(x,x+NMAX,res,f);
```

The "Map" template produces granules by splitting large "transform" into lesser ones, which constitutes library-supplied grains of parallelism. It should be noted, that, since aggregation is done at compile time, individual `do_something` calls may be inlined by compiler inside the granule loop. This enables all toolset of optimizations, that are available for loops in modern C++ compilers.

6 Future work

It's clear, that for the T-system, runtime overhead may be incurred not only from the task and thread management, but from the variable mechanism as well. Consider the following naive program to calculate N-th prime number:

```
// n -- desired prime number
// j -- current number
// i -- number of primes found <=j
tfun long nprimes (int n, long j, tval long i) {
    tval long tmp;
    tval long ni;
    tmp = nprimes (n,++j,ni); // start the

    bool is = is_prime(++j); //verify, if the number is prime

    if (n==i) return j; // Runtime environment should cancel
                        // subsequent "nprimes" calls started,
    else {
        if (is) {
            ni = ++i;  // increment the number of primes found
            return tmp; // return the result of subsequent calls
        }
        else {
            ni = i; // no change, connect non-ready
            return tmp;
        }
    }
}
```

For this program to be executed effectively in parallel, runtime system should provide "lazy" task evaluation strategy, as well as an ability to cancel tasks, which result are not necessary. The first is existing, and the latter is a prospective feature of OpenTS. In principle, the overhead of thread management may be kept low with the help of "inlining" technique [3]. However, management of non-ready variables `ni` and `tmp` may claim more CPU cycles than useful `is_prime`, especially, in OpenTS, where grabage collector is present. One of future work directions may be investigation of dynamic specialization mechanism for non-ready variables.

7 Conclusion

Implemntation of granules aggregation technique improves a lot ease of use for parallel programming tool. "The program mer takes on the burden of identifying what can be computed safely in parallel, leaving the decision of exactly how the division will take place to the run-time system." [3] The runtime support library may vary the "weight" of tasks in wide limits, so it capable to adapt program to wide variety of parallel computers that exist today: multi-core, SMPs, computational clusters with different kind of interconnects. At the same time, programmer may write very simple code, separating the computation code from the code, managing computational process (scheduling, aggregation and so forth). However, development of adaptive mechanisms, capable to measure individual granule weight and aggregate them accordingly, is a subject of future work, as well as attempt to provide lightweight non-ready variables.

Acknowledgments. This work is supported by Russian Foundation of Basic Research grant N 050708005ofi_a and basic research program of Presidium of Russian Academy of Science "Development of basics for implementation of distributed scientific informational-computational environment on GRID technologies".

References

1. Chandra R., Menon R., Dagum L., Kohr D., Maydan D., McDonald J.: Parallel Programming in OpenMP Morgan Kaufmann 2000
2. Schloegel K.,Karypis G.,Kumar V.: Graph Partitioning for High-Performance Scientific Simulations in Dongarra J. et. al: "Sourcebook of parallel computing", Morgan Kaufmann, 2003
3. Kranz D., Halstead R., Mohr E.: Mul-T,A High-Performance Parallel Lisp ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, OR, June 1989, pp. 81–90
4. Mohr E., Kranz D., Halstead R.: Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs IEEE Trans. Parallel Distrib. Syst., **2**,3(1991),264-280,

5. Wagner D., Calder B.: Leapfrogging: a portable technique for implementing efficient futures Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming
6. Halstead R,: MULTILISP: a language for concurrent symbolic computation ACM Transactions on Programming Languages and Systems (TOPLAS), **7**,4 (1985),501–538
7. Zhang L., Krintz C., Soman S.: Efficient Support of Fine-grained Futures in Java International Conference on Parallel and Distributed Computing Systems (PDCS), November 2006, Dallas, TX
8. Loidl H-W.,Trinder P.W.,Butz C.: Tuning Task Granularity and Data Locality of Data Parallel GpH Programs Parallel Processing Letters, **11**,4 (2001) 471–486.
9. Loidl H-W.: Granularity in Large-Scale Parallel Functional Programming PhD. Thesis. University of Glasgow. March 1998. Available online: `http://www.dcs.gla.ac.uk/~hwloidl/publications/PhD.ps.gz`
10. Alt M.: Coding Considerations Practical Methods to Maximum Efficiency for Intel Itanium Architecture Intel Corp., 2004
11. Abramov S., Adamovich A. I., Inyukhin A., Moskovsky A., Roganov V., Shevchuk E., Shevchuk Yu., Vodomerov A.: OpenTS: An Outline of Dynamic Parallelization Approach. Parallel Computing Technologies (PaCT)-2005, Krasnoyarsk, Russia, September 2005, LNCS **3606** (2005) 303-312.
12. Talia D.: Advances in Programming Languages for Parallel Computing in Annual Review of Scalable Computing (2000)//Yuen C. K. 28–58
13. Stroustrup B.: The Design and Evolution of C++ Addison-Wesley, 2004 (in Russian translation: Piter, St.Petersburg, 2007)
14. Intel Thread Building Blocks, `http://www.intel.com/cd/software/products/asmo-na/eng/294797.htm`, Intel Corp.
15. An P. et. al: STAPL: An Adaptive, Generic Parallel C++ Library Wkshp. on Lang. and Comp. for Par. Comp. (LCPC), Cumberland Falls, Kentucky, Aug 2001, pp 193–208.
16. Bischof H., Gorlatch S.,Leshchinskiy R. DatTel: A Data-parallel C++ Template Library HLPP 2003: Second International Workshop on High-Level Parallel Programming and Applications, 15–17 June 2003, Paris, France .
17. Chiba S.: A Metaobject Protocol for C++ Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), pp 285–299, October 1995
18. McNamara B., Smaragdakis Y.: Functional Programming in C++ The 2000 International Conference on Functional Programming (ICFP), Montreal, Canada, 18–20 September 2000
19. Abelson H.,Sussman G.J.: Structure and Interpretation of Computer Programs The MIT Press, 1996.
20. Randall K.H.: Cilk: Efficient Multithreaded Computing. Ph. D. Thesis, MIT Department of Electrical Engineering and Computer Science.June 1998.
21. Luszczek P., Bailey D., Dongarra J., Kepner J., Lucas R., Rabenseifner R., Takahashi D.: The HPC Challenge (HPCC) Benchmark Suite SC06 Conference Tutorial, IEEE, US, Tampa, Florida, November 12, 2006.
22. Kuszmaul B. A Cilk Response to the HPC Challenge (Class 2) SC06 Conference, IEEE, USA, Tampa, Florida, November, 13-16, 2006.