The High Performance Numerical Computing on Service-Oriented Architectures Project: Research Themes

Peter Strazdins,
(and the HPNumSOA Project team),
Department of Computer Science,
The Australian National University

seminar at Platform (TM) Computing, Toronto, 21 October 2008

(slides available from http://cs.anu.edu.au/~Peter.Strazdins/seminars)

THE AUSTRALIAN NATIONAL UNIVERSITY

1 Overview

- project overview
- report to date
- overview of (parallel) numerical applications
- approaches to extending Symphony for numerical applications
- conjugate-gradient iterative linear system solver in Symphony
- LINPACK: blocked algorithm, message passing implementation
- data services: Global Arrays, GemFire
- parallel LINPACK under a service-oriented paradigm
  - 'persisting service tasks' and 'service sub-tasks' formulations
- conclusions: extending Symphony for numerical applications

2 Project Overview

- collaborative research project between Departments of Computer Science at ANU & Adelaide and Platform Computing, from July 2007
- funded by ARC Linkage Grant and Platform (TM) Computing (+ DCS ANU)
  - funding structure inherited from a previous collaborator (withdrew 03/08)
- CIs Peter Strazdins (ANU) & Paul Coddington (Adelaide)
- Platform staff include Khalid Ahmed, Chris Smith & Jingwen Wang
- PhD scholar Jaison Mulerikkal
  - project must be scoped within a single PhD candidature!
- aim: to utilize service-oriented infrastructures (Symphony) to develop HPC applications on clusters / grids
  - develop programming models that extend shared memory paradigms to clusters / grids
  - characterize the applications that run run well on service-oriented infrastructures
  - investigate the programmability and optimization of such applications
  - devise a performance modelling methodology (use for run-time decisions

3 Project Report (to date)

- July 2007: before & after (and after again!): recruiting of scholars
- Sep 2007: both scholars to be at ANU; 1st scholar starts
- Nov 2007: 1st scholar terminates!! 2nd starts
- Mar 2008: decide to continue as a 1-scholar project
- Jan-Jul 2008:
  - identify conjugate-gradient solver as a key application to develop
  - identify ‘globally-addressable’ data service required for LINPACK
  - specific training for Jaison; begin literature review
  - deploy Symphony (source / object) & develop (naive) CG solver
- Aug-Oct 2008:
  - Global Arrays & GemFire considered for data service (LINPACK)
  - 64-bit installation of Symphony from source succeeded; deployment
  - preliminary performance results on naive (and improved) CG program on DCS clusters

4 Overview of (Scientific) Numerical Applications)

- embarrassingly parallel: e.g. Monte Carlo methods, parametric simulations
- matrix-(matrix/vector) multiply-based (sparse or dense):
  - iterative linear and eigensolvers (e.g. conjugate-gradient)
  - matrix-matrix: quantum scattering, neural net training
- dense linear algebra computations
  - e.g. LINPACK, least-squares, symmetric eigensolver
- regular grid computations, molecular dynamics and (naive) particle simulations
- computational fluid dynamics (e.g. NAS Parallel Benchmarks)
- complex semi-regular / irregular
  - e.g sparse matrix factorizations, multigrid, fast-multiple methods

5 Extending Symphony for Numerical Applications

- service tasks receive input (also common data) from client; send output back to client for aggregation
- multiple service tasks may run on on a single service instance process
  - possibly large overhead per setting up an SI, but potentially low overhead for individual service task
  - requires computation granularity large enough to mitigate cost of moving data
- opportunity for fault-tolerance (restart uncompleted tasks on failed SI) and load balancing (send more tasks to am SI that finishes early)
  - a simple solution to the problem of heterogeneity!
  - wish to retain relative simplicity of programming in any extensions
- initial idea: service tasks communicate via OpenMP or MPI
  - OpenMP slow! loss of load-balancing and fault-tolerance
  - what is role of Symphony anyway?
  - are there more service-oriented approaches?

THE AUSTRALIAN NATIONAL UNIVERSITY

6 Conjugate-gradient Linear System Solver in Symphony

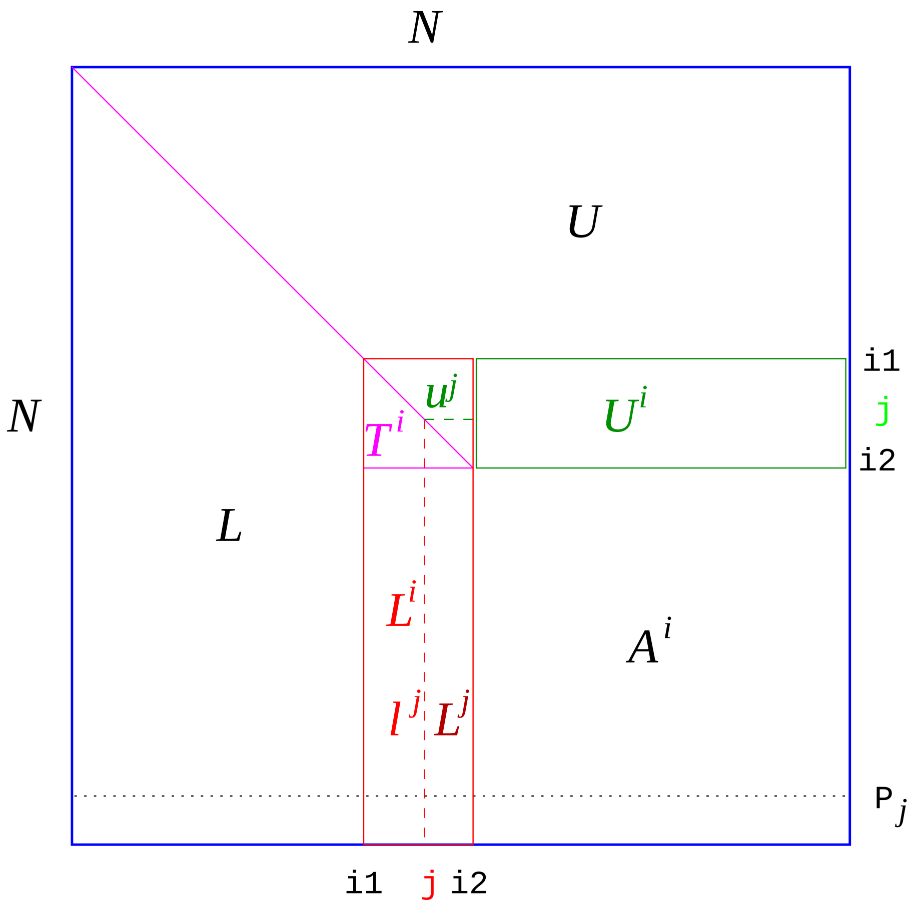
- solve positive definite symmetric linear system $Ax = b$, A is $N \times N$
- repeatedly compute $w = Au$ ($O(N^2)$), using a ‘batch’ of service tasks; plus a number of ($O(N)$) vector-vector operations (on client task)
  - w from output of service tasks; what are their inputs, common data?
- a parallelization strategy ($p = 3$ tasks): $A \times u = w$
- number of iterations r is generally largish, but slowly grows with N
- can avoid communicating A on each iteration if whole computation is performed in a single Symphony session with $r \times p$ service tasks
  - the whole of A is common data
    - slightly wasteful, as each SI may not need all of A
  - for each ‘generation’ of service tasks, u could be regarded as ‘secondary’ common data

7 Parallel LINPACK: Introduction

- general dense linear system solution of $Ax = b$ via LU factorization with partial pivoting
  - $A \rightarrow LU$, A is $N \times N$
- used in many apps.; LINPACK is the #1 scientific benchmark
- represents a whole class of dense linear algebra computations
  - use of blocked algorithms to optimize memory hierarchy performance
    - from registers to interconnects!
  - both right -looking and left-looking variants
  - symmetric matrix computations have some extra challenges

8 Blocked (Right-looking) LU Factorization Algorithm

- uses Gaussian elimination with partial pivoting on an $N \times N$ matrix A



$i_1 = i\omega, i_2 = i_1 + \omega$

for (j=i_1; i<i_2; j++)

  find $P[j]$ s.t. $|L^i{}_{P[j],j}| \geq |L^i{}_{j:M,j}|$

  $L^i{}_{j,:} \leftrightarrow L^i{}_{P[j],:}$

  $l_j \leftarrow l_j / L^i_{j,j}$

  $L_j \leftarrow L_j - l_j u_j$

for (j=i_1; j<i_2; j++)

  $A_{j,:} \leftrightarrow A_{P[j],:}$ (outside L^i)

(row broadcast of T^i, L^i)

$U^i \leftarrow (T^i)^{-1} U^i$

(column broadcast of U^i)

$A^i \leftarrow A^i - L^i U^i$

9 Parallel LINPACK: Message Passing Implementations

- basic algorithm is efficient for the bulk of the computation (update trailing sub-matrix)
- uses a $P \times Q$ logical processor grid (minimize communication volume) and the block-cyclic matrix distribution (load balance)
- has $O(N \lg P)$ messages, $O(N^2)$ communication volume, $O(N^3)$ FLOPs
  - thus inherently scalable (for large enough N)
- panel formation (L^i, U^i)
  - basic technique: storage blocking ((ScALAPACK)
  - advanced techniques (load-imbalance / communication overhead trade-off):
    - lookahead: portable HPL (UTK, 2000)
    - algorithmic blocking (ANU, 1995–2000)
    - panel scattering / PBMD (UTexas, ANU, 1998–2000)
- these implementations assume homogeneity and are not fault-tolerant!

10 Parallel LINPACK: the Block-cyclic Matrix Distribution

- 'standard' for most parallel dense linear algebra applications
  - good load balance for tri. & sub- matrices; r affects performance
- divide matrix A into $r \times s$ blocks on a $P \times Q$ processor grid; block (i, j) of A is on proc. $(i\%P, j\%Q)$
- eg. if $r = 3, s = 2$ on a 2×3 grid, a 10×10 matrix A is distributed:

a_{00}	a_{01}	a_{06}	a_{07}	a_{02}	a_{03}	a_{08}	a_{09}	a_{04}	a_{05}
a_{10}	a_{11}	a_{16}	a_{17}	a_{12}	a_{13}	a_{18}	a_{19}	a_{14}	a_{15}
a_{20}	a_{21}	a_{26}	a_{27}	a_{22}	a_{23}	a_{28}	a_{29}	a_{24}	a_{25}
a_{60}	a_{61}	a_{66}	a_{67}	a_{62}	a_{63}	a_{68}	a_{69}	a_{64}	a_{65}
a_{70}	a_{71}	a_{76}	a_{77}	a_{72}	a_{73}	a_{78}	a_{79}	a_{74}	a_{75}
a_{80}	a_{81}	a_{86}	a_{87}	a_{82}	a_{83}	a_{88}	a_{89}	a_{84}	a_{85}
a_{30}	a_{31}	a_{36}	a_{37}	a_{32}	a_{33}	a_{38}	a_{39}	a_{34}	a_{35}
a_{40}	a_{41}	a_{46}	a_{47}	a_{42}	a_{43}	a_{48}	a_{49}	a_{44}	a_{45}
a_{50}	a_{51}	a_{56}	a_{57}	a_{52}	a_{53}	a_{58}	a_{59}	a_{54}	a_{55}
a_{90}	a_{91}	a_{96}	a_{97}	a_{92}	a_{93}	a_{98}	a_{99}	a_{94}	a_{95}

11 Candidate Data Services: Global Arrays

- Global Arrays is an MPI-compatible library supporting access to a logically global array
  - parallel tasks specify a (blocked) multi-dimensional array distribution
  - initialize contiguous sections in a local array, and then put this into a global array
  - from then, other tasks can access via get operations into local array, and and put (or accumulate-put) sections back into global storage
  - communications are one-sided; regular barriers are required
- potentially some performance disadvantages over MPI implementations using collective operations
  - partially mitigated by synchronous versions of get/put
- current implementation has no support for fault-tolerance
- a wide range of applications can be implemented in this, but:
- only *rumours* of support for block-cyclic distribution
  - row or column replicated sub-arrays problematic

12 Candidate Data Services: Data Fabrics

- GemFire is a general distributed data fabric
- support for various modes of replication
- in-core caching for fast access if data is local
- already some experience in use in conjunction with Symphony
- support for get / put in array sections? block-cyclic?
  - however, linear algebra communication libraries provide this (via a general matrix copy function)
  - *in principle*, should not be too difficult to integrate this into GemFire

13 Parallel LINPACK under a Service-oriented Paradigm

- assume a suitable data service exists; client initially pushes A to the service
  - `Get()`, $\texttt{Get}_H$`()`, $\texttt{Get}_V$`()`, $\texttt{Get}_{HV}$`()`:
    get all, horiz. portion, vert. portion, horiz./vert. portion of sub-matrix
- service tasks progress in *phases*:
  - pull new / out-of-date matrix sections perform updates, and push updated matrix sections
  - this would effectively be a commit operation (synchronization point)
  - if a task fails to reach the point (in reasonable time), it could be restarted elsewhere
- issues:
  - if $O(N)$ phases are required, is this too much overhead?
  - how to write the application so that it is restartable at any phase?
  - retaining load-balancing ability (in heterogeneous environment)

14 Parallel LINPACK: Persisting Service Tasks

```
for (j=i₁; i<i₂; j++)
  Get_V(l^j)
  find local P[j] s.t. |L^i_{P[j],j}| ≥ |L^i_{j:M,j}|
  Put-MaxAccum(P[j])
  Get(P[j], v = l^j_{P[j]}) (scalars)
  if task owns row j:
    Get(L^i_{j,:}, L^i_{P[j],:})
    perform local swap;
    Put(L^i_{j,:}, L^i_{P[j],:})
  if tasks owns column j:
    l_j ← l_j/v
    Put(l_j) (V)
  Get_H(l_j); Get_V(u_j); Get_HV(L_j)
  L_j ← L_j − l_j u_j
  Put(L_j)
```

```
for (j=i₁; j<i₂; j++)
  Get_HV(A_{j,:}, A_{P[j]}) (outside L^i)
perform local swap;
for (j=i₁; j<i₂; j++)
  Put(A_{j,:}, A_{P[j]}) (outside L^i)
Get(T^i); Get_V(U^i)
U^i ← (T^i)^{-1}
Put(U^i)
Get_V(U^i); Get_H(L^i); Get_HV(A^i)
A^i ← A^i − L^i U^i
Put(A^i)
```

15 Parallel LINPACK: Persisting Service Tasks

- assumes the panels (L^i,U^i) are only distributed along their length when they are being formed (storage blocking, panel scatting)
- performance issue: need smart `Get()`, which does not-reload up-to-date data already loaded
- fault-tolerance: system maintains implicit counter for implicit synchronization points (`Put(); Get()`)
  - for restarted tasks, `Get/Put()` calls are no-operations until start point reached
  - also, local computation code blocks should be skipped
- the number of SI is a factor of logical task grid $P \times Q$
  - some affinity should be retained, i.e. each SI gets a sub-grid of $P \times Q$ (performance)
  - load balancing: at (major) synchronization points, if an SI appears consistently overloaded, tasks may be stopped & restarted elsewhere
  - left-looking variants will reduce volume of data from Put()s

16 Parallel LINPACK: Service Sub-task Formulation

- *client tasks* orchestrates $O(NPQ)$) a series of *different* get-compute-put service tasks

  `for (j=`i_1`; i<`i_2`; j++)`

  Determine $P[j]$ from l^j

  Get l^j, $P[j]$, $A_{j,:}$, $A_{P[j]}$ & to do swap & scale l^j

  Get l_j, u_j & L_j to update L_j

  Do Row Swaps of $A_{j,:}$, $A_{P[j]}$ (outside L^i)

  Get T^i & U^i to update U^i

  Get L^i, U^i & A^i to update A^i

- fault-tolerance and load-balancing much easier to implement
- need however extensions to enable multiple versions of `invoke()`
  - some common control information (e.g. j) needs also to be passed
  - stresses more heavily the client–SM–SSM–SI chain
  - programming a series of sub-tasks is arguably a little more tedious
  - affinity in that corresponding tasks in successive 'generations' are scheduled (all other things being equal) on the same SI

17 Conclusions: Extending Symphony for Numerical Apps

- it appears possible to do so for dense linear algebra applications
  - service-oriented paradigm can be maintained, together with potential for fault-tolerance
  - a suitable, highly efficient and fault-tolerant data service is required
    - none seems to currently exist
    - some intelligence require, i.e. optimize redundant portions in `get()`s
- changes required in Symphony
  - notion of 'generations' of service tasks
  - extension to mechanism of common data
  - inside SIs, we also need to cache data to ensure fast Get()s
- issue: can all of this be made robust, lean and efficient enough?
- for performance, may consider using less frequent synchronization points
- if successful for dense linear algebra, likely to be successful to other challenging applications (except complex irregular)